THE DATA-DRIVEN MULTITHREADING VIRTUAL MACHINE

Samer Arandi

University of Cyprus, 2011

Since the advent of digital computers, chip designers built faster computers by relying on improvements in fabrication technologies and architectural/organizational optimizations. However, the inability of the sequential model to tolerate long latencies (manifested mainly in the Memory Wall) combined with the Power and Instruction Level Parallelism (ILP) Walls problems eventually rendered this approach ineffective.

The envisaged solution was to switch to multi-core architectures. This switch was an engineering effort that did not address the long memory latencies and the complexity of the designs. In addition, this switch elevated concurrency as a major challenge as it became evident that new concurrent models/paradigms are needed to efficiently utilize the resources of multi-core chips.

The Data-flow model is a formal model that can handle concurrency and tolerate memory and synchronization latencies. Data-Flow systems can also be simpler and thus more power efficient than conventional systems.

In this thesis, we propose re-visiting the Data-flow model and adopting it as the basis for an execution model that efficiently exploits the resources of multi-core architectures. We design and implement a virtual machine supporting the Data-Driven Multithreading (DDM) model of execution, which combines Dynamic Data-Flow concurrency with efficient sequential execution, on multi-core architectures. We refer to this virtual machine as the Data-Driven Multithreading Virtual Machine (DDM-VM). The DDM-VM also supports execution on a cluster of multi-core

nodes. We develop a number of alternative approaches facilitating the programming of the DDM-VM, in addition to a number of optimizations for improving the performance.

The evaluation of the DDM-VM for both single nodes and clusters demonstrates that the VM scales well and tolerates latencies and synchronization overheads efficiently, achieving very good performance and outperforming other similar state-of-the-art systems.

The main contribution of this thesis is the design, implementation and optimization of a virtual machine that efficiently exploits data-flow concurrency on commercial multi-cores and delivers high-performance comparing favorably with similar systems. The rest of this thesis contributions are:

- The development of the DDM-VM, an efficient virtual machine that supports DDM execution on multi-core architectures. It utilizes Data-Flow concurrency for scheduling threads and efficient sequential execution within a thread, while optimizing the context management of the Dynamic Data-Flow tagging system. The virtual machine has two individually optimized implementations: The DDM-VM$_s$ tailored for homogeneous multi-cores and the DDM-VM$_c$ tailored for heterogeneous multi-cores. The latter is developed for heterogeneous multi-core architectures with a host/accelerator organization and a software-managed memory hierarchy. The DDM-VM$_c$ is also a high-performance implementation of DDM. When comparing its performance with two similar state-of-the-art systems using a number of computationally-intensive benchmarks, the DDM-VM$_c$ outperforms both systems and achieves 88% of the theoretical peak performance for one of the benchmarks. For the same benchmark the DDM-VM$_c$ execution on a cluster of 4 machines achieves 0.44 TFLOPs.

- The development of a fully-automated software prefetching cache with variable cache block sizes and explicit data locality optimizations for handling explicitly-managed memory hierarchies.

- The development of the support for distributed DDM execution. The DDM-VM is the first DDM implementation supporting distributed DDM execution across a cluster of multi-core nodes.

- The development of the support for runtime dependency resolution using specialized I-Structures. The DDM-VM is the first DDM implementation that supports parallel execution of code that contains producer-consumer dependencies that are only resolved at runtime while utilizing compile-time resolution at the same time. This permits taking advantage of the strengths of both approaches and expands the class of programs that can be mapped to the DDM model. It also has the potential to improve the programmability and enhance the yield of compilation methods generating data-flow code.

- The development of a number of performance optimizations and monitoring & visualization tools.

THE DATA-DRIVEN MULTITHREADING VIRTUAL MACHINE

Samer Arandi

A Doctor of Philosophy Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

at the

University of Cyprus

Recommended for Acceptance

by the Department of Computer Science

November, 2011

2011

APPROVAL PAGE

Doctor of Philosophy Dissertation

THE DATA-DRIVEN MULTITHREADING VIRTUAL MACHINE

Presented by

Samer Arandi

Research Supervisor _____
Professor Paraskevas Evripidou

Committee Chair _____
Dr. Pedro Trancoso

Committee Member _____
Professor Ian Watson

Committee Member _____
Dr. Yiannakis Sazeides

Committee Member _____
Dr. Costas Kyriacou

University of Cyprus

November, 2011

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor Professor Skevos Evripidou for his continuous support for me both personally and professionally. I am grateful for his patience, guidance and immense knowledge and experience. He has taught me various valuable things through out the period of my studies and made sure I am always on the right track with his deep intuitions and infallible advice. It was a privilege and a pleasure having him as my advisor.

I would also like to warmly thank Dr. Pedro Trancoso for his invaluable help and support since the first day of my studies. Thanks for the sincere encouragement, inspiring opinions and insightful discussions.

It is a pleasure to extend my gratitude to the rest of the committee members: Professor Ian Watson, Dr Yannakis Sazeides and Dr. Costas Kyriacou for their encouragement and insightful comments.

I would also like to thank all my colleagues whom I had the pleasure to work and collaborate with: Thank you George Michael, Andreas Diavastos, Petros Panayi and Kyriacos Stavrou.

I gratefully acknowledge the financial support of the Cyprus Research Promotion Foundation which made my Ph.D. work possible.

There are so many people whose support and friendship I have enjoyed during the course of my Ph.D. I would like to sincerely thank my friends: Sandy Ayas, George Michael, Maria Efthemio, Maria Charalambous, Nuno Martins, Petros Panagyi and Kyriacos Stavrou. Thanks for helping me during my stay in Cyprus and for making my time here a very enjoyable one.

Last but not least, I would like to thank my family. My parents: Dr. Najeh Arandi and Samar Arandi for their unconditional love, support and -most importantly- patience, during the past 5 years and for all that they did for me throughout my life, which I cannot reward them back for, no matter what I do or how hard I try. Whatever I am today, its because of them. I would also like to thank my younger brother and sister Ayman and Samah for their love and encouragement and my little nieces Gazal and Masa for all the happiness they brought into our lives.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Introduction

Since the advent of digital computers, in the early 1940s, the computer architecture field has been dominated by the sequential model of execution. Advocates of parallel processing have being predicting the end of sequential computing and the shift to parallel processing [12] since the 1960s. However, chip designers have been using the exponentially increasing number of transistors (predicated by Moore's Law) to postpone the shift indefinitely, by designing more complex processors with larger cache sizes and continuously increasing the clock frequency.

However, the inability of the sequential model to tolerate long latencies (manifested mainly in the Memory Wall problem [132] referring to the widening gap in performance between the processor and the memory) combined with the Power Wall [129] and the Instruction Level Parallelism (ILP) Wall [129] problems eventually rendered this approach ineffective. The Power Wall refers to the prohibitive increase in power consumption and generated heat resulting from the complexity of the designs, and the ILP Wall refers to the scarce degree of exploitable Instruction Level Parallelism (targeted by such designs). Figure 1 shows the slow down in performance gains due to these problems.

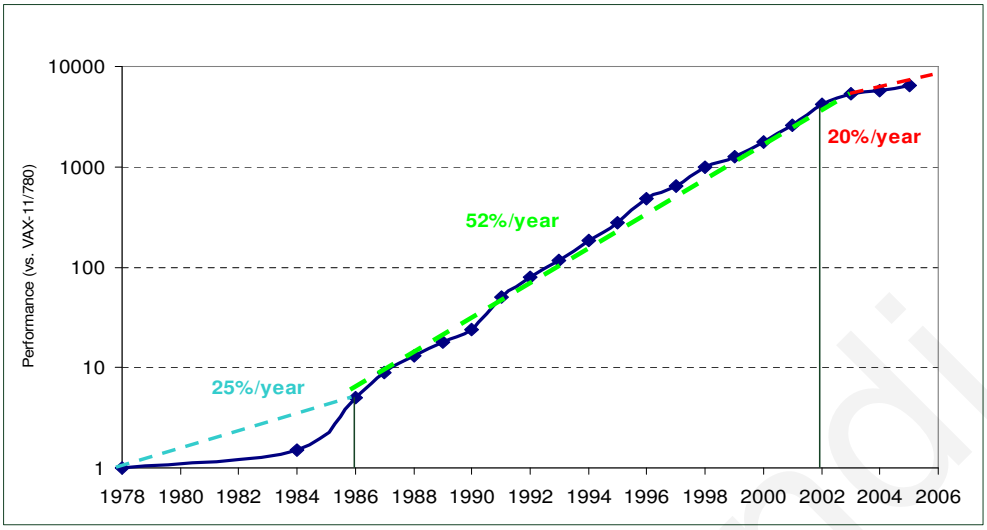


Figure 1: Growth in processor performance since the mid-1980 relative to the VAX 11/780 measured by the SPECint benchmark (Figure from Hennessy & Patterson [93])

The envisaged solution was to utilize lower frequencies and use the silicon state to pack more cores on a chip [90]. These chips are known as Chip-Multiprocessors (CMP) or Multi-core Processors. Multi-core processors can be either homogeneous, consisting of similar cores or heterogeneous, consisting of cores with different properties. The motivation behind heterogeneous design is two folds [69]: First, it results in a more efficient utilization of the cores through better adaptation to the diversity of applications. The second advantage comes from a more efficient use of the die area for a given thread-level parallelism, which allows for a more power and area efficient design.

1.2 Motivation

The switch to multi-cores was an engineering effort that did not address the fundamental issues that caused the previously stated problems: the long memory latencies (it actually exacerbates it as we will show shortly) and the complexity of the designs (Out-of-Order execution and large caches) that caused the power wall.

Moreover, this switch elevated *concurrency* as a major issue in utilizing the increasing number of cores on a single-chip, as it soon became evident that traditional programming and execution models do not allow for efficient utilization of the large number of resources now available on a single chip. This task is even more complex on heterogeneous multi-core architectures, as different types of resources need to be individually optimized in order to achieve maximum global performance.

Another challenge facing the new multi-core designs is the Memory Wall, as even with lower clock rates, the problem manifests itself due to the increasing number of cores compared to the available resources dedicated for memory on the chip. This gets worse as the number of cores on the die increases [85]. One of the techniques applied to combat the memory wall is to utilize fast explicitly-managed on-chip memories in addition to the slower off-chip memory in the system. This technique is utilized by various stream processors such as the Stanford Merrimac [32] and Imagine [65] processors and modern parallel architectures such as the NVIDIA G80 [89] and the IBM Cell/B.E [63].

Using explicitly-managed memory hierarchies offers a great opportunity for increasing the performance by efficiently utilizing the memory resources, but at the same time poses a considerable challenge, as it typically requires the programmer to explicitly manage all data transfers between on-chip and off-chip memories, manage data allocation in the on-chip local memories, and guarantee a coherent view of the data. This further aggravates the problem of utilizing concurrency on multi-core architectures.

The Data-flow model [33, 11, 131] is a formal model that can handle concurrency in a distributed manner and tolerate memory and synchronization latencies efficiently, since an operation in data-flow is scheduled to execute when its data is ready. Moreover, the semantics of data-flow avoids the need for synchronization constructs like locks, barriers and busy-waiting. Dynamic

Data-flow can expose the maximum degree of parallelism in a program as it only enforces *true dependencies*. Thus, the Data-flow model does not suffer from the limitations of the sequential model, mainly, the inability to tolerate memory latencies. Furthermore, Data-Flow systems can be simpler and more power efficient than conventional systems. Consequently, data-flow based models are a competitive candidate as the execution models for exploiting the resources of multi-core architectures.

1.3 Problem Statement - Hypothesis

In this thesis, we try to explore a Data-Flow based execution model for the efficient utilization of the resources of multi-core architectures.

1.4 Approach

We follow an evolutional path by utilizing Dynamic Data-Flow on conventional Multi-core systems. We advance the state-of-the art of the Data-Driven Multithreading (DDM) model, which combines the distributed concurrency of the Data-Flow model with the efficient execution of the Control-Flow model.

The goal of this thesis is the design, implementation and optimization of a virtual machine supporting the DDM model of execution on multi-core systems. The virtual machine has two implementations, the first is designed for homogeneous multi-cores and the second is designed for heterogeneous multi-cores with a host/accelerator organization that utilize a software-managed memory hierarchy. A representative example of such architectures is the powerful Cell/B.E. processor. The virtual machine also supports distributed DDM execution on a cluster of multi-core nodes.

We summarize the goals of the proposed virtual machine:

- The implementation of a data-driven execution model on homogeneous and heterogeneous multi-cores (intra-node DDM).
- The implementation of automatic management of software-managed memory hierarchies through the development of a prefetching software cache utilizing data-driven caching policies.
- Supporting data-driven execution across distributed multi-core nodes (inter-node DDM).

The proposed virtual machine hides the low-level details of the parallel resources of the underlying machine and uses a unified representation for DDM programs. The VM, composed of the Thread Scheduling Unit (TSU) and the supporting runtime, handles the tasks of thread scheduling, synchronization and execution instantiation implicitly. A special prefetching software cache based on data-driven caching policies is developed to handle software-managed memory hierarchies.

We provide a number of alternative approaches for programming the virtual machine. The programmer can use a set of C macros that expand into calls to the VM or utilize a number of compiler *directives* with the aid of a preprocessor tool. Two other compilation tools are under development. The first utilizes the GCC compiler and the second utilizes the CnC declarative parallel programming language. We also provide monitoring and debugging tools to help with application development. A suite of 10 benchmarks was ported and used for performance evaluation and comparison with state-of-the-art systems.

1.5 Thesis Contributions

The main contribution of this thesis is the design, implementation and optimization of a virtual machine that *efficiently* exploits Data-Flow concurrency on conventional/commercial multi-cores and *outperforms* other existing systems. The rest of the contributions include:

- **Contribution 1**: Development of the Data-Driven Multithreading Virtual Machine (DDM-VM), an efficient virtual machine that supports Data-Driven Multithreading execution on homogeneous multi-cores and heterogeneous high-performance multi-core systems. The DDM-VM utilizes DDM scheduling for exploiting the resources of multi-core architectures and tolerating synchronization and memory latencies. The DDM-VM has the following properties:
  - It adheres to the formal dynamic data-flow constructs as described by the U-Interpreter.
  - It has two individually optimized implementations: The DDM-VM$_s$ tailored for homogeneous multi-cores and the DDM-VM$_c$ tailored for heterogeneous multi-cores.
  - The DDM-VM$_c$ is a high-performance implementation of DDM that achieves better performance than similar state-of-the-art systems. It is also the first DDM implementation optimized for heterogeneous multi-core architectures with a host/accelerator organization and a software-managed memory hierarchy.
- **Contribution 2**: Development of Software CacheFlow (S-CacheFlow), a fully-automated software prefetching cache with variable cache block sizes and explicit data locality optimizations for handling explicitly-managed memory hierarchies. Two implementations of the S-CacheFlow are developed and evaluated. The implementation that distributes part the data transfer tasks to the cores (distributed S-CacheFlow) is adopted as the default due to its performance advantage.
- **Contribution 3**: Development of the support for distributed DDM execution. The DDM-VM is the first DDM implementation supporting distributed DDM execution across a cluster of multi-core nodes. Previous implementation either supported distributed DDM execution across single-processor nodes [73] or DDM execution within a multi-core node [113].

- **Contribution 4**: Development of the support for runtime dependency resolution using specialized I-Structures. The DDM-VM is the first DDM implementation that supports parallel execution of code that contains producer-consumer dependencies that are only resolved at runtime. The developed approach introduces a *helper/proxy* thread that resolves such dependencies at runtime and updates the consumer(s) accordingly with the help of an *I-Structure*. This permits taking advantage of the strengths of both compile-time and run-time dependency resolution simultaneously and expands the class of programs that can be mapped to the DDM model. It also has the potential to improve the programmability and enhance the yield of compilation methods generating data-flow code.

- **Contribution 5**: Development of a number of performance optimizations and monitoring & visualization tools.

Programming both implementations of the VM is done via a unified programming interface -utilizing C macros that hides most of the low-level differences of the underlying architectures. This interface is the target of a number of tools that facilitate the programming of the DDM-VM and provide various alternative approaches.

1.6 Performance Evaluation

1.6.1 Methodology

We evaluate the two implementations of the DDM-VM for both single-node and distributed multi-node execution. The benchmark suite used in the evaluation consists of ten applications featuring kernels widely used in scientific and image processing applications. All of the benchmarks are coded in C using the DDM-VM *macros* and compiled using the compilers available from the IBM Cell SDK V2.1 in the case of the DDM-VM$_c$ implementation and the GCC 4.4.3 compiler in

the case of the DDM-VM$_s$. The DDM-VM$_c$ implementation runs on a Sony Playstation 3 (PS3) machine with Linux. For the evaluation of the distributed execution we used a cluster of 4 PS3 machines.

The DDM-VM$_s$ implementation runs on a 12-core machine composed of two six-core AMD Opteron processors. For the evaluation of the distributed execution we use two clusters (to test different configurations of nodes/cores-per-node):

1. The first is composed of two 12-core machines (*System-1* cluster)
2. The second is composed of four 4-core machines (*System-2* cluster)

1.6.2 Results

1.6.2.1 DDM-VM$_c$ Evaluation

- **Optimizations Evaluation:** To evaluate the effects of resource management and locality and synchronization memory optimizations on the performance, we use the MatMult and Cholesky benchmarks as two case studies. The first application is a representative of applications with a simple dependency graph and the second is a representative of applications with a complex dependency graph. Moreover, both applications are computationally intensive and performance-sensitive. The result of this evaluation is used to guide the performance optimization for all the benchmarks in the rest of the evaluation.
  - **Effect of Resource Management:** To assess the DDM-VM resource management control mechanisms (both at the TSU level and at the program level) we have executed two sets of experiments for both benchmarks. In the first, we have varied the size of the TSU's Extended Firing Queue (ExFQ) and in the second, we have utilized *Loop Throttling* and varied the limit on the number of concurrent invocations of the

throttled threads. The results show that both mechanisms are effective in controlling the concurrency.

- **Effect of Locality Optimizations:** We compare the performance of the benchmarks with and without the locality optimization. The results demonstrate that utilizing locality improves the performance for both applications. The main source of improvement is the reduced demand of the private Local Store (LS) memory space, which permits fitting the data of more threads, thus allowing the TSU better chance to prefetch data and overlap latencies with computation. This result demonstrates the deep implications the size of the LS memory has on the execution behavior and consequently the importance of taking into account the size of the working set when choosing the granularity of the threads.
- **Effect of Synchronization Memory Organization:** As the operation of the Synchronization Memory is critical for the performance of DDM execution, we evaluated 3 different SM implementations. The results show that the *direct* implementation, which preallocates the SM entries achieves the best performance. The *associative* implementation, which allocates the entries on demand performs 2nd best on average. The performance of the *hybrid* implementation, which attempts to conserve the allocations by re-using SM entries depends on the execution pattern (locality of the SM updates) of the executed application.

- **General Performance Evaluation:**

  - **Effect of Thread Granularity and Software CacheFlow Implementations:**

Thread Granularities

The results show that the performance improves as the threads granularity increases. As higher granularities amortize better the scheduling overheads of the TSU and S-CacheFlow operations and -further- allow DDM-VM$_c$ to hide the latency of data transfers through prefetching/multi-buffering.

S-CacheFlow Implementations

The distributed S-CacheFlow implementation (which distributes part the data transfer tasks to the SPE cores) performs better than the basic S-CacheFlow on all of the benchmarks in general. The advantage of the distributed implementation is clear when the number of cores is higher, as it precludes the PPE from becoming a bottleneck due to the demand of the S-CacheFlow.

– **Effect of Problem Size:**

To assess the effect of the program size on performance we have executed the benchmarks for different problem sizes. The results show that the system generally scales well across the range of the benchmarks achieving almost linear speedup for the large problem sizes, as large problem sizes result in longer execution time, which amortizes initialization and parallelization overheads.

– **Concurrency and Latency Tolerance:** To evaluate the potential of the DDM-VM$_c$ in exploiting concurrency and tolerating synchronization and memory latencies, we have performed a number of experiments in which we limit the number of threads that can be scheduled concurrently to 1 (purely sequential scheduling of DDM-VM applications), 2 and 3. We compare the results with a normal (non-DDM) sequential

program. The results show that enabling the scheduling and execution of multiple concurrent threads permits the TSU to overlap the scheduling and data transfers latencies (via prefetching) with the execution of the threads. This illustrates that DDM-VM$_c$ effectively leverages the decoupling of synchronization and computation for maximum tolerance of latencies.

– **GFLOPs Performance and Comparison:** To examine the efficiency of the DDM-VM$_c$ we report the GFLOPs performance results of three computationally intensive applications, MatMult, Cholesky and Conv2D and compare them with the CellSs [18, 94] and Sequoia[41] platforms that target the Cell processor.

  * The results show that MatMult and Conv2D scale almost linearly for the large problem size and achieved 132 GFLOPs (88% of the theoretical peak performance) and 77 GFLOPs, respectively on 6 SPEs. The Cholesky application achieves a speedup of 5 out of 6 despite its complex dependency graph, yielding 101 GFLOPs for the large problem size.
  * Comparing the performance with CellSs for MatMult and Cholesky, DDM-VM$_c$ achieves an average of 42% and 112% performance improvement for MatMult and Cholesky, respectively. Moreover, DDM-VM$_c$ achieves the best improvement v.s. CellSs for the smaller problem sizes, which indicates that it introduces less overhead for exploiting concurrency.
  * Comparing the performance with Sequoia for MatMult and Conv2D, DDM-VM$_c$ achieves an average of 12% and 25% performance improvement for MatMult and Conv2D, respectively.

The results indicates the efficiency of the DDM-VM$_c$ and its ability to outperform other platforms on the Cell.

- **Distributed DDM-VM$_c$ Execution:** The results of evaluating the distributed DDM-VM$_c$ execution on a four PS3 cluster shows that:
  - The system achieves an average of 80% of the maximum possible speedup when utilizing various number of SPEs per node for all the benchmarks on the largest input size.
  - As the input size increases the system scales better: the average speedup (on all the benchmarks) utilizing all the SPEs is 13.4 out of 24 for the smaller input size and 16.54 out of 24 for the larger input size. This is expected as larger problem sizes allow for amortizing the overheads of the parallelization.
  - Compared to single-node execution larger input sizes and larger granularities are needed in general for the system to scale due to the additional latencies introduced by the network data and synchronization messages transfer.
  - When utilizing all the SPEs on the four nodes the system delivers an impressive 0.44 TFLOPs for the MatMult benchmark and 178 GFLOPs for the Conv2D benchmark (the two computationally intensive benchmarks), which demonstrates the efficiency of the distributed execution on the DDM-VM$_c$.

1.6.2.2 DDM-VM$_s$ Results

Overall, the results of the DDM-VM$_s$ evaluation confirm the findings of the DDM-VM$_c$ evaluation:

- **Effect of Thread Granularity:** When executing the benchmarks with varying thread granularities, the results show that the performance improves as the granularity increases, since higher granularities amortize better the scheduling overheads of the TSU.

- **Effect of Input Size:** When executing the benchmarks for various problem sizes. The performance improves as the input size increases, since larger problem sizes result in longer execution time, which amortizes initialization and parallelization overheads.

- **Overall Performance:** The results of executing all the benchmarks demonstrate that overall, the system scales well over the range of the benchmarks and achieves - when utilizing all the cores - an average speedup of 9.6 out of 11 (the maximum possible speedup is 11 since we reserve one core out of the 12 for the execution of the TSU), which indicates the efficiency and scalability of the system.

- **Runtime Dependency Resolution Evaluation:** We evaluate our technique for handling runtime-determined dependencies by studying the effect of the overheads of the I-Structure operations on the performance. We compare the performance of 3 versions of a subset of the benchmarks for various thread granularities. The first version utilizes the compile-time approach for resolving the dependencies. The second version combines both approaches and the third utilizes the runtime approach.

  The results demonstrate that:

  - The best performance is delivered by the version utilizing the compile-time approach, followed by the one utilizing the combination of the compile-time and runtime approaches.

  - The performance loss (relative to the compile-time version) is higher for lower granularities and decreases as we increase the granularity. For example, when using 10 cores in one of the applications, the performance loss when utilizing the runtime approach for all the dependencies is 43% for the smallest granularity compared to 13.6% for the largest granularity. When utilizing a combination of the two approaches the loss is 14.8% for the smallest granularity compared to 2.2% for the largest granularity. The same observation applies to the rest of the benchmarks.
  - Utilizing run-time dependency resolution (for part or all of the data dependencies in the evaluated programs) achieves acceptable performance compared to the compile-time approach, whilst utilizing thread granularities in the range we normally utilize in DDM-VM programs.

- **Distributed DDM-VM$_s$ Execution:** The results of evaluating the distributed DDM-VM$_s$ execution on the *System-1* and *System-2* clusters confirm the findings of the distributed DDM-VM$_c$ evaluation:

  - The system achieves an average of 80% and 84% of the maximum possible speedup when utilizing various number of cores per node for the largest input size on *System-1* and *System-2* clusters, respectively.
  - The system scales better as the input size increases.
  - Larger input sizes and granularities (compared to single-node execution) are needed for the system to scale.

1.7 Thesis Outline

In Chapter 2 we present background information followed by a review of the related work. Chapter 3 presents the architecture of the DDM-VM and its two implementations. The chapter also describes the design and implementation of the prefetching software cache utilized for handling software-managed memory hierarchies. In Chapter 4 we describe the support for distributed DDM execution across a cluster of multi-core nodes. Chapter 5 presents the programming methodology and tool-chain utilized with the DDM-VM, in addition to a number of optimizations employed to improve the performance of the DDM-VM. In Chapter 6 we describe the support for runtime dependency resolution. The evaluation results for the two DDM-VM implementations for both single-node execution and distributed execution are presented in Chapter 7. Finally, the conclusion and future work are presented in Chapter 8.

Chapter 2

Background and Related Work

2.1 Introduction

In this chapter we present background information and related work focusing on multi-core architectures and data-flow. For multi-core architectures, we follow the evolution of computer architectures from monolithic to multi-core. The debut of the multi-core architectures and the reasons behind it are then presented. We highlight heterogeneous multi-core designs and present the motivation behind this approach and the advantages it promises in respect to power and area efficiency compared to the homogeneous designs. We also illustrate some of the current heterogeneous processors and systems, focusing on the Cell/B.E. heterogeneous processor. We illustrate some of its unique features like the explicitly-managed private memories of its cores.

Following that, we review the data-flow model proposed in the early 70's by Jack Dennis and highlight its strengths and weaknesses. We follow its evolution from the static & dynamic data-flow architectures into the hybrid data-flow/control-flow and multithreading architectures. Finally, we review state-of-art work related to the topic of this thesis.

2.2 Background Information

2.2.1 From Monolithic to Multi-core Architectures

The trend in the last two decades for achieving high-performance was driven by the increase in the operating frequency and extracting more instruction-level parallelism (ILP) by exploiting sophisticated hardware techniques like Out-of-Order & Superscalar execution [118, 110, 49]. This was feasible due to the exponentially increasing number of transistor predicted by Moore's Law [84]. This trend enabled the engineers to come up with very complex designs and use large sizes of cache. Although successful for several years this strategy hit many walls: the memory, power, complexity and ILP walls [132, 90, 129]. We present more details on these issues in the subsequent sections.

2.2.1.1 Pipelining Processors

With the success of the Reduced Instruction Set Computer (RISC) architectures in the early 80's, most of the RISC machines utilized pipelining to improve performance. Pipelining exploits parallelism between instructions (Instruction Level Parallelism - ILP) to increases the instruction throughput, which translates into a reduction in the total execution time of the application. However, the ideal potential of pipelining is hardly achieved [93] due to imbalances in the time of the different pipeline stages, the overhead introduced by pipelining and the different pipeline hazards that result from limited hardware resources (structural hazards) or the properties of the executed program (data and control hazards). Figure 2 illustrates the state of a five-stage pipeline with the ideal execution of seven instructions.

To improve the performance of pipelining many techniques, both software and hardware, are deployed. *Resource duplication* and *functional unit pipelining* are utilized to reduce structural

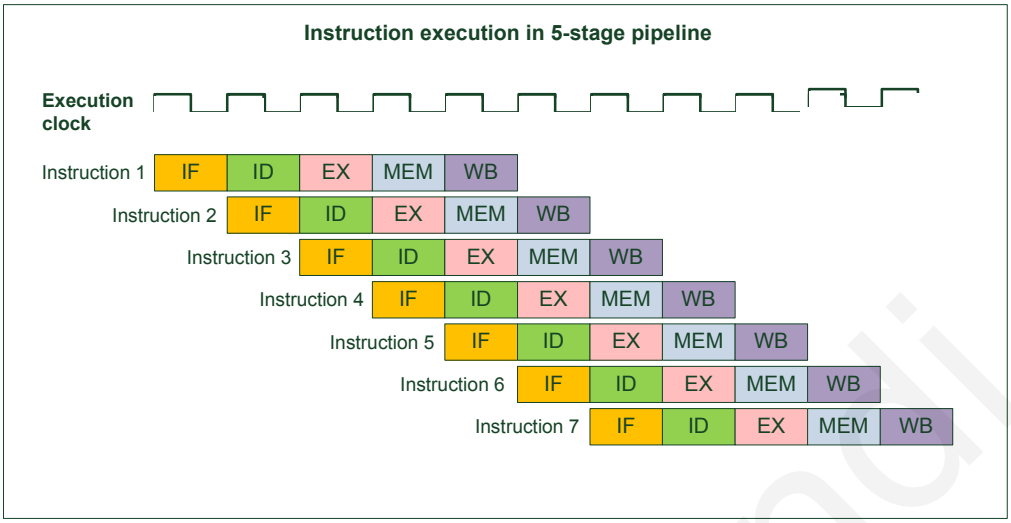


Figure 2: A five-stage pipeline with ideal execution of seven instructions

hazards. *Forwarding* and *Software Scheduling* are utilized to reduce data hazards. The former, forwards the results of some functional units as direct inputs for other units in the pipeline to avoid a stall. The latter -utilized by the compiler- re-arranges instructions so as to increase the distance between dependent instructions. The compiler can also schedule instructions to reduce the effect of branches (the source of control hazards) by filling the branch delay slot usefully which avoids the stall that would have been needed waiting for the result of the branch to be calculated.

Branch prediction is utilized by the compiler to reduce control hazards. This involves static prediction of branches where the branch is always assumed to be taken or not-taken and then a restart of the pipeline if the prediction result was wrong. Most of the previous techniques are utilized with *unrolling* which also reduces the number of branches in a loop and consequently the number of control hazards.

Due to the limited accuracy of static prediction most of the recent processors utilize *dynamic branch predication* techniques implemented in hardware. In its simplest form dynamic prediction employs a branch history table that records the results of a branch and indexes it by hashing the

address of the branch instruction. More sophisticated schemes are also utilized which correlate the results of other recent (nearby) branches as well.

2.2.1.2 Dynamic Scheduling

Dynamic scheduling (or out-of-order execution) allows the hardware to re-arrange the order of instruction execution to reduce the effects of hazards without breaking the data dependencies between the instructions of the program. This scheme has the advantage that it allows the processor to tolerate events like a cache-miss by executing other non-dependent instructions while waiting for the data to arrive. It also can handle the cases where the compiler cannot reason about the dependency at compile time. As dynamic scheduling introduces the potential for name dependencies: Write-After-Read and Write-After-Write (WAR and WAW), *register renaming* is utilized to handle this issue by renaming all destination registers including those of pending read or write instructions referring to an earlier instruction.

2.2.1.3 Hardware-based Speculation

Hardware-based speculation is a technique that reduces the effect of control hazards further than what branch predication can do. It speculates the outcome of branches and executes the program as if the speculation guess was correct. To support the ability to recover the state of the processor in case the speculation was not correct, instructions are allowed to finish execution however, committing their results to the register file is delayed until after the outcome of the branch is determined.

2.2.1.4 Multi-issue Architectures

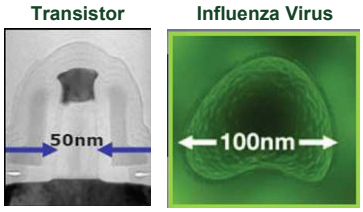
Multiple-issue processors try to improve performance by issuing more than one instruction per cycle. These processors can be classified into two types:

1. Statically/Dynamically Scheduled Superscalar Processors
2. Very Long Instruction Word (VLIW) Processors

The first issues multiple numbers of instructions per cycle, while the second issues a fixed number of instructions packed into one *Mega* instruction per cycle. Both the static issue superscalars and the VLIW processors rely on the compiler for scheduling the program instructions. Dynamically scheduled superscalars add hardware to the issue and commit logic of the pipeline to support multiple issue and multiple commit of instructions per cycle.

2.2.1.5 Power and ILP Walls

As more transistors continuously became available to designers more aggressive wider-issue superscalars were built. The transistors were also used to increase the size of the caches and the clock frequency was continually increased. Figure 3 depicts a table of the number of the transistors available on a chip every year including a prediction for the years to come. This provided an incremental increase in the performance for a decade, however, this increased the complexity of the designs and demanded more power, increased the leakage problems and generated more heat. This lead to higher costs for thermal packaging, fans, electricity and air conditioning. Moreover, higher-power systems increase the chances of failures [69]. Wall has shown in a seminal paper [129] that the degree of ILP rarely exceeds seven even assuming the most ambitious hardware configuration for a superscalar (this is especially true for integer programs). The envisaged solution for these problems was to use the silicon estate to scale the number of cores on chip by putting



	2004	2006	2008	2010	2012	2014	2016	2018
Feature Size	90nm	65nm	45nm	32nm	22nm	16nm	11nm	8nm
Billions Transis	2	4	8	16	32	64	128	256

Figure 3: A comparison between the sizes of a 50nm transistor and the Influenza virus. Table on the right shows the number of transistors available/predicted on a chip up to 2018 (information from Intel).

more processors on the same die [90] and shift the focus to extracting Thread Level Parallelism (TLP).

2.2.1.6 Chip Multiprocessors

Microprocessors incorporating more than one processor on the same chip are known as Chip-Multiprocessors (CMP) or Multi-core Processors. One way to classify multi-cores is as Homogeneous and Heterogeneous (or asymmetric) multi-cores. Exampels of homogeneous multi-cores include Intel's Core 2 Duo [57] and Core 2 Quad [58], AMD's Athlon 64 X2 [59] and Turion 64 X2 [60], IBM's Power5[64], and SUN's T1 Niagara[67]. All these processors are homogeneous multi-core processors with two to eight cores per chip. Currently, the parallelism offered by these processors is mostly exploited for *throughput computing*. While current multi-core designs achieve acceptable speedups partially due to the effective shared secondary cache, these designs are not expected to scale to many cores. Intel has made available a new experimental 48-core SCC processor [54] and demonstrated another proof of concept 80-core processor [61]. This ushers the beginning of what is called by many as the Many-Core era.

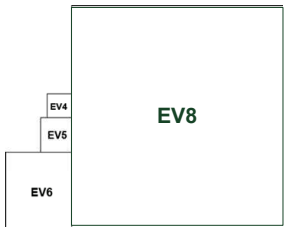
2.2.1.7 The Memory Wall

The dramatic difference between the exponential rate of improvement in microprocessor speed and the exponential rate of improvement in memory speed, in favour of the former, and the consequent result this has on the performance of computer system is commonly referred to as the Memory Wall. This problem is one of the well-studied fundamental problems in computer architecture [132, 82].

The move to multi-core architectures did not solve this problem, as even with lower clock rates, the memory wall manifests itself due to the increasing number of cores compared to the available resources dedicated for memory channels. The constrained resources (power and area budget) limit the the number of available independent channels into memory (which affects latency), and the speed and width of those channels (which affects bandwidth). This gets worse as the number of cores on the die increases [85].

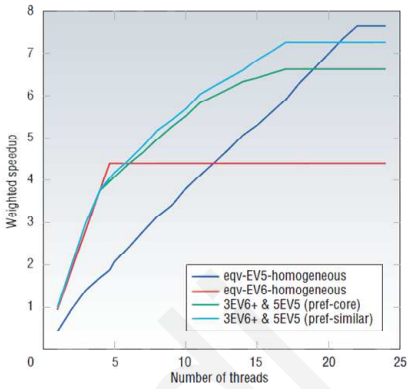
2.2.1.8 Heterogeneous Multi-cores

Heterogenous multi-cores offer advantages over homogenous multi-cores in areas of Power and Throughput and allow for mitigating the effects of Amdahl's Law [69]. Different applications require different sets of resources. Indeed, it is very common that the very same application requires different resources during different phases of its execution. Heterogeneous multi-cores provide a better match for the varying requirements & characteristics of applications, which leads to an efficient utilization of the size of the chip and its power consumption. The parts of applications requiring the power-hungry, complex out-of-order execution with branch-predication and sophisticated speculation techniques, run on the cores suitable for that. The other parts requiring less control-flow and more computational and data processing capabilities run on other types of cores, for example, cores with SIMD capabilities. It has been shown that using heterogeneous



Core	Peak Power	Average Power	Performance (normalized IPC)
EV4	4.97	3.73	1.0
EV5	9.83	6.88	1.3
EV6	17.8	10.68	1.87
EV8	92.88	46.44	2.14

(a)



(b)

Figure 4: (a) Area and Power comparison for four generations of the alpha processor (b) Performance comparisons for a number of heterogeneous & homogeneous configurations (figure and data from [69])

multi-cores improves energy efficiency per instruction by four to six times [48]. Executing the serial portions of a program, which are the portions limiting the amount of parallelism according to Amdalh's Law, on the fast but relatively area and power-inefficient core, and the parallel portions on a larger number of smaller simpler cores, can maximize the ratio of performance to power dissipation [69]. Figure 4-a depicts a comparison in the area and power budget of four generations of the alpha processor. Figure 4-b demonstrates the primary benefit of heterogeneity as it shows that the heterogeneous architecture configuration generally provides the highest performance across all levels of thread parallelism. An analytical evaluation of the effect of Amdahl's law on homogeneous and heterogeneous multi-cores [53] demonstrated the potential of heterogeneous multi-cores in achieving speedups much greater than what homogeneous multi-cores can obtain.

It is relevant to note that the previous discussion of heterogeneous multi-cores focuses on tightly coupled heterogeneous multi-cores, which could be seen as a point in a wider spectrum of heterogeneous processors that vary in the degree of coupling between the cores. Some of the processors have the cores tightly coupled on the same chip (like the Cell/B.E. processor [63]), or

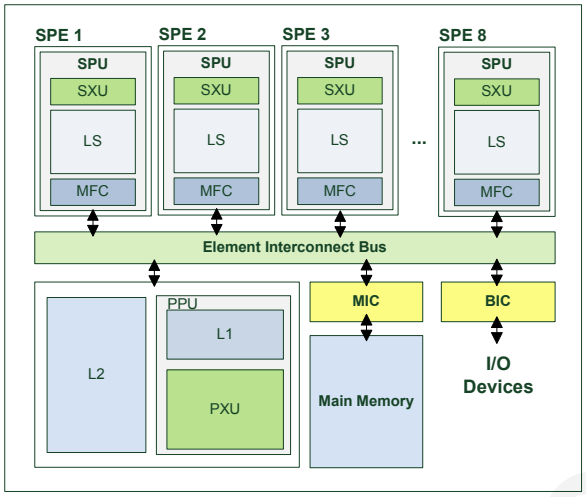


Figure 5: The Cell Processor Architecture

on the same board like the onboard graphic cards (and old FPU accelerator), or communicating through I/O as in modern desktops equipped with powerful graphic cards like the nVidia G80 [89] series, or as in the Clearspeed [55] and the Imagine Stream processor [7] boards, or even coupling the cores across the network as in various Visualization Clusters. Heterogeneous processors application domains include numerous fields like HPC, image & video processing, medical imaging, gaming & graphics, weather forecasting, financial, cryptographic and network processing. In the following section we focus on the architecture of the Cell/B.E. processor, a heterogeneous multi-core processor developed by IBM, Sony and Toshiba.

2.2.1.9 The Cell/B.E. Heterogeneous Multi-core

The Cell Broadband Engine processor (Cell/B.E. or Cell for short) [63] is a heterogeneous multi-core chip composed of one general-purpose RISC processor called the Power Processor Element (PPE) and eight fully-functional SIMD co-processors called the Synergistic Processor Elements (SPE) communicating through a high-speed ring bus called the Element Interconnect Bus (EIB). Figure 5 illustrates the architecture of the Cell processor.

The PPE has two levels of cache and is designed to run the operating system and act as a coordinator for the other cores (SPEs) in the system. The SPE is a RISC processor with 128-bit SIMD organization that is capable of delivering 25.6 GFLOPs in single-precision. It has its own 256KB software-controlled local store (LS) memory. The SPE can only execute instructions and access data existing in its LS. The data has to be explicitly fetched by the programmer from main memory via the asynchronous Direct Memory Access (DMA) engine of each SPE's Memory Flow Controller (MFC) unit.

The Address Space and Communication Infrastructure

The Cell architecture defines two separate memory address spaces: the main memory shared address space and the Local Store private address space. The general purpose PPE core accesses the main memory address space normally through the L1 & L2 hardware cache hierarchy. On the other hand, the SPE cores can only execute code and access data that reside in their private LS memory.

The communication between the two address spaces is managed explicitly by the programmer using software via DMA calls. In addition to the DMA calls that is used for large granularity communication (128 byte to 32 MB per message), the Cell provides small granularity communication mechanisms (1 to 4 32-bit messages) called *mailboxes* and *signals*.

The Cell processor design adopts a weakly consistent data storage model that allows storage accesses to be re-ordered dynamically [5], which provides an opportunity for improved overall performance and reduced memory latency. However, this requires that programs explicitly order accesses to storage if they want stores to occur in the program order. This is achieved using special *fencing* and *barrier* instructions.

Programming the Cell Processor

A typical Cell program consists of two sets of source files, one set for the part of the program running on the PPE and another set for the part running on the SPEs. Both sets of files are compiled separately with different compilers and at link-time all the binaries of the SPEs are embedded inside the PPE binary. At run-time the programmer code in the PPE program creates a specialized *pthread* for each SPE and loads the image of the SPE binary into the corresponding *pthread* and starts its execution on the SPE. The programmer is required to explicitly manage the allocation of data in the constraint LS and handle the data-communication between the LS of the SPEs and the main memory, in addition to the synchronization tasks. Moreover, the limited size of the LS demands efficient utilization

Discussion

The Cell is a high-performance processor with nine cores that is capable of delivering 256 GFLOPs. However, harnessing its power is not trivial. A suitable execution model that efficiently exploits its resources and accommodates its unique properties is required. The model must address the heterogeneity of its nine cores and its novel architecture elements, especially its software-controlled memory hierarchy and communication infrastructure.

2.2.2 Data-flow Architectures

The data-flow model is a formal model of execution that was proposed by Jack Dennis [33] in the early 70's as an alternative to the control-flow model. This model is functional (computations have no sides effects) and asynchronous (the only condition for executability is data availability). The advantages of the Data-flow model is its distributed concurrency control -as there is no central point of control- and its ability to expose the maximum inherent parallelism in a program, as it inflicts the least constraints on execution, i.e., the only condition for an instruction to start executing is the availability of its input data. These advantages allow Data-flow architectures to tolerate the synchronization and memory latencies and extract more parallelism compared to traditional control-flow architectures.

Programs in the data-flow model may be represented as a graph where nodes represent instructions and arcs represent data paths. Each instruction has an opcode, slots for holding operands values, and the destination instructions address field(s). Tokens carry data values and travel along the arcs from the producer instruction to the consumer one(s). When all the operands of an instruction are present the instruction is enabled and -thus- deemed executable.

2.2.2.1 Static and Dynamic Data-flow Architectures

Data-flow architectures can be classified according to the way the model handles storage-per-arc or re-entrance. Static Data-flow Architectures or Single-Token-Per-Arc (proposed by Dennis [34]) allow -at most- one token to be present at any arc, while Dynamic Data-flow Architectures (proposed by Arvind et al. [13, 11, 14] and Watson & Gurd [130]) allow more than a token to be present, and so, use a tag to identify the logical position of the token in the arc.

Despite the promise of exploiting more parallelism than Static Data-flow Architectures, Dynamic Data-flow Architectures suffered from a number of shortcomings: token-matching proved

to be a very expensive operation given the fine-grain level data-flow works at. The Associative memory that would be ideally used for that purpose proved infeasible and the use of the more practical hashing techniques was not fast enough. The Explicit Token Store (ETS) was developed within the Monsoon data-flow processor [92] to eliminate the need for the associative memory by allocating a separate memory frame (called activation frame) for each activation of a loop or re-entrant code block. The activation frame holds the synchronization information of instructions within the code block. As access to locations within the activation frame is performed through offsets relative to a pointer to that frame, there is no need for associative memory searching. The ETS principle was also applied in other data-flow machines like the Eplilon-2 [47] where it was called Direct Matching.

2.2.2.2 Hybrid Data-flow Architectures

Despite the proposed Explicit Token Store/Direct Matching technique, the per-instruction token-matching and fine-grained context switching at the level of each instruction, which made it not possible to use registers, resulted in poor performance when executing sequential code compared to conventional control-flow architectures. These problems in addition to the inefficient handling of data structures (like arrays) prevented pure data-flow architectures from delivering the expected performance gains. To address these problems a number of proposals emerged where data-flow and control-flow architectures converged to form hybrid architectures [75] utilizing techniques from both principles. One of the ways to classify these hybrid architectures is as: Threaded data-flow [92, 47, 104], Large-grain data-flow [88, 29, 28, 79, 86, 40] or RISC data-flow [109].

2.2.2.3 Multithreaded Architectures

One of the roots of multithreading is the data-flow model [117], as combining the instruction-level context switching with sequential scheduling can be seen as an evolution of the hybrid data-flow architectures towards Multithreading. In particular, Threaded data-flow and Large-grain data-flow can be classified as non-blocking multithreaded architectures [108], since data-flow principles are utilized to start the execution of the non-blocking threads. And once a thread starts executing it will execute to completions without suffering any long memory or synchronization latencies.

To describe the different multithreading architectures we will use the classification adopted by [108, 117]. Multithreading Architectures can be classified as *implicit* or *explicit*. Implicit architectures refer to ones that can execute several concurrent threads from a single sequential program. This can be done with the help of a compiler (statically) or dynamically by the hardware. Implicit multithreading is applied to increase the performance of a single program thread. Examples of Implicit Architectures include the Multiscalar Processors [111, 128], the Trace Processors [101, 126], the Speculative Multithreaded Processor [80, 121] and the Speculative Data-Driven Multithreading (SDDM) [102].

Explicit Multithreading Architectures, on the other hand, refer to architectures that execute threads of the same or different processes concurrently. These architectures aim at increasing the performance of a multiprogramming workload (sometimes on the expense of single thread performance). Examples of Explicit Multithreading Architectures are Interleaved, Blocked and Simultaneous Multithreading (SMT) Architectures.

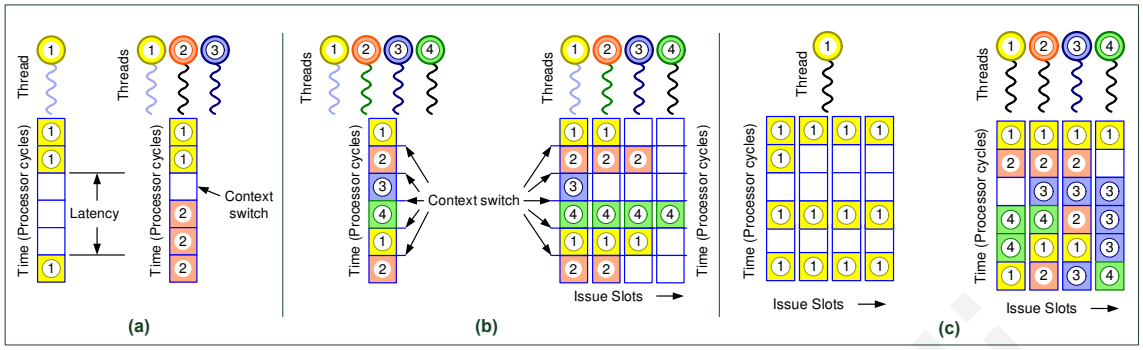


Figure 6: Different multithreading approaches (a) Blocked Multithreading (b) Interleaved Multithreading (c) Simultaneous Multithreading (SMT)

Blocked and Interleaved Multithreading:

The distinction between Blocking and non-Blocking architectures derives from the notion of blocking v.s non-blocking threads. A non-blocking thread executes until completion without interruption, i.e., without blocking the processor pipeline due to remote memory, cache misses or synchronization latencies. This is achieved by starting the execution of the thread only when all input operands are ready. Thus a program is compiled into (relatively small) threads activating each other in a producer-consumer manner. Threaded Data-flow and Large-grain Data-flow architectures are examples of non-blocking multithreading architectures. Examples of processor utilizing the Blocking Multithreading approach are, the MIT Sparcle processor [6] and the Rhamma processor [50]. In Interleaved Architectures on every cycle an instruction is fetched from a different thread and fed into the pipeline of the processor, thus the processor performs a context switch after executing each instruction. This approach tolerates memory latency, however, it comes at the expense of degrading single-thread performance. Examples of well-known interleaved multithreaded processors include the Heterogeneous Element Processor (HEP) [74], the Cray Multithreaded Architecture (MTA) and the SB-PRAM prototype processor.

Simultaneous Multithreading

The Simultaneous Multithreading (SMT) approach combines multithreading with superscalar techniques by providing several hardware contexts and register sets on the processor and issuing instructions from several threads simultaneously. This allows tolerating the latencies when executing one thread by issuing instructions from the other threads. Note that interleaving and blocked multithreading techniques are most efficient when applied to scalar RISC or VLIW processor [117]. On the other hand, SMT is more efficient with superscalar architectures as it takes advantage of ILP and TLP at the same time. This allows SMT to achieve superior performance for multi-threaded/multi-program applications as the TLP can be exploited from threads of the same applications or threads from different applications. Figure 6-c illustrates an example of the SMT approach.

Two early SMT processors are The SMT processor proposed at the University of Washington [124, 38, 123] and the Multithreaded Superscalar Processor [125, 107] from the University of Karlsruhe. Examples of commercial processors utilizing the SMT technology include the (cancelled) Alpha 214614 (EV8) [35], Intel *Hyperthreading* [122], IBM Power5 [64] and Intel Nehalem.

2.2.2.4 Recent Data-flow Projects

Many of the techniques that originated in the Data-flow computational model have found their way into modern processor architectures (e.g. out-of-order execution [93] and non-blocking threads) and compiler technologies (e.g. Single Static Assignments (SSA) [31] and register renaming). A brief overview of some of the recent projects related to Data-flow is given next. In Section 2.3 we review state-of-the-art work that is directly related to the work of this thesis.

Scheduled Data-flow

Scheduled Data-flow (SDF) [66] is a non-blocking decoupled memory/execution multithreaded architecture. A program in SDF is compiled and partitioned into non-blocking computation threads and memory-access threads. Data of the computation threads is preloaded into enabled register contexts prior to execution and results are post-stored from the registers into memory after the execution completes. Instructions within a computation thread adhere to the single-assignment rules of data-flow albeit using a conventional control-flow like sequencing. A separate unit, called Synchronization Pipeline (SP), handles pre-load and post-store operations, while execution is undertaken by the Execution Pipeline (EP). A simple Scheduled Unit (SU) handles the tasks of scheduling threads and allocating the corresponding memory frames, synchronization counters and register sets.

Decoupled Threaded Architecture - Clustered

The Decoupled Threaded Architecture - Clustered (DTA-C) [44] is an architecture that is based on the SDF architecture [66] with the addition of the concept of *clusterizing* resources. As the name implies the architecture is composed of a set of clusters or tiles. Internally, each cluster consists of one or more Processing Elements (PEs) and a Distributed Scheduler Element (DSE). The set of all DSEs constitutes the Distributed Scheduler (DS) responsible for assigning threads at runtime. All clusters are connected via a complex inter-cluster network. Elements within a cluster are connected via a faster, less complex intra-cluster network. Each processing element contains pipelines, frame memory, register file and local scheduler.

As in SDF each thread has a unique *continuation* and a frame which holds its data. However, scheduling of threads is done on two levels: among the PEs within the cluster and across clusters.

The Local Scheduler (inside each PE) and the Distributed Scheduler are responsible for assigning continuations and frames to the threads and for keeping track of processor usage in order to balance the load in the system.

The EDGE Architectures

The Explicit Data Graph Execution Architecture (EDGE) [24] proposes a new ISA that supports direct instruction communication that expresses the data-flow graph the compiler generates. This allows the hardware to deliver the output of a producer instruction directly as an input to consumer instruction(s) rather than writing it back to a shared namespace (memory or temporary registers). Thus, exposing a higher degree of concurrency and achieving a more power-efficient execution as no complex hardware is needed to discover data-dependencies dynamically at run-time. A program is partitioned by the compiler into *hyper-blocks* comprising a large number of instructions. An instruction doesn't encode its source operands, but rather, the compiler specifies the location of where the produced results will be routed according to the consumer instructions. Each hyperblock is executed atomically in parallel on an array of functional units (ALUs).

The WaveScalar Architecture

The WaveScalar [116, 115] is a data-flow ISA and execution model designed for scalable low-complexity/high-performance processors. It is composed of a large number of processing nodes surrounded by intelligent cache banks that hold the current working set of instructions. Instructions execute in-place and explicitly communicate with its dependent instructions in a data-flow fashion. The WaveScalar compiler breaks the control-flow graph of a program into single-entrance directed acyclic blocks of instructions called waves (example of a wave is a loop iteration). Each

wave is tagged via a distributed tagging mechanism using special instructions to differentiate between different dynamic instances of a wave.

Fuce: The Continuation-based Multithreading Processor

The Fuce processor [8] is based on the data-flow-like continuation-based multithreading model, which is optimized for the execution of Thread Level Parallelism (TLP). A Thread is defined as a block of instructions that work on registers (except for loads and stores) and execute without interruption until completion. Each thread is associated with a synchronization counter which is decremented upon receiving special events called continuations (initiated by other threads). When the counter reaches zero the thread becomes ready for execution. The processor comprises multiple Thread Execution Units (TEU) and one Thread Activation Controller (TAC), which controls the scheduling of the threads.

Programs are written in a high-level programming language called CML, which is based on C and is extended with a set of special words and thread-related instructions.

2.3 Related Work

In this section we review work directly related to the topic of this thesis. We mainly focus on state-of-the-art work.

2.3.1 Threaded Abstract Machine (TAM)

The Threaded Abstract Machine (TAM) [28] is a parallel self-scheduling execution model that is based on Data-Flow. A TAM program consists of a collection of blocks, where each block consists of several non-blocking threads that enable other threads and generate asynchronous messages. Each block is associated with an *activation frame* that provides storage for the local variables in addition to the resources required for the synchronization and scheduling of threads. Data and control dependencies are enforced using synchronization counters within the frame. A *synchronizing thread* is associated with an *entry_count* that is decremented by threads *forks* and *posts*. The thread is enabled when the count reaches zero. The entry_count value, thread forks and posts are all controlled by the compiler. The TAM employs a threaded machine language (called TL0) and implements a compilation path from the Id90 programming language to TL0. The TL0 code is used as a machine independent intermediate form that can be used to generate C in addition to other targets.

The data-driven scheduling approach employed by the DDM model is similar to that of the TAM. However, as noted in [71], the main difference between the two is that in the case of the TAM thread synchronization and scheduling are controlled entirely by the compiler and directly carried out by the code of the threads, however, in the case of the DDM model (and consequently the DDM-VM) these operations are carried out by the Thread Scheduling Unit.

2.3.2 Star Superscalar (StarSs)

Star Superscalar (StarSs) [18, 94, 96] is a parallel programming platform that targets symmetric multiprocessors and multi-cores, the Cell processor and GPUs. It schedules annotated tasks at run-time based on data-dependencies. StarSs focuses on the ease of programmability and portability and utilizes a source-to-source compiler and a number of runtime libraries. Unlike the approach adopted by our work, where we build the dependency graph statically if possible, StarSs always builds its task dependency graph at run-time. This approach incurs extra overheads as it resolves the dependencies at run-time even if they can be resolved at compile-time. Moreover, this approach makes only part of the dependency graph available to the scheduler and consequently a fraction of the concurrency opportunities in the applications is visible at any time. In section 6.1 we demonstrate how the DDM-VM utilizes both compile-time and run-time dependency resolution to gain the benefits of both approaches. Performance comparisons with the Cell implementation of the StarSs platform is found in the Evaluation Chapter of this thesis.

2.3.3 Sequoia

Sequoia [41] is a programming language that facilitates the development of memory hierarchy aware parallel programs. It provides a source-to-source compiler and a runtime system for multi-cores and clusters of multi-cores including the Cell processor. Unlike our proposed approach, however, Sequoia requires the programmer to use special language constructs and types and focuses on portability. It also uses a hierarchical/recursive task execution paradigm rather than one based on data-dependencies. Performance comparisons with the Cell implementation of this platform is found in the Evaluation Chapter of this thesis.

2.3.4 Concurrent Collections

Concurrent Collections [22, 23] is a high-level parallel programming language that is based on the *separation of concerns* concept. It allows domain experts who have deep expertise in their respective domain but lack knowledge in parallel programming to express their programs in high-level declarative constructs. A tuning expert who has knowledge in parallel programming can optimize these programs for the underlying target machine.

A program is described declaratively in terms of computational *steps* that communicate via data *items* that satisfy the dynamic single assignment property. *Steps* and *items* are uniquely identified by *tags*. The code of the steps can be written using imperative languages like C or java.

Unlike DDM that has data-dependence relationships only, CnC represents both data-dependence and control-dependence. Furthermore, resolving the dependencies in CnC is performed at runtime, while in the DDM-VM the dependency resolution is performed at compile-time and if this is not possible, it can be also performed at run-time as will be described in Section 6.1. Furthermore, in the Future Work Chapter we describe a preliminary effort for a CnC to DDM compiler, which is motivated by the the matching between DDM and CnC.

2.3.5 Open Multi-Processing (OpenMP)

Open Multi-Processing (OpenMP) [2] is a widely-utilized parallel programming API standard that supports shared-memory programming on multiple platforms. It consists of a run-time library and a set of compiler directives utilized to identify sections of code that can be parallelized. OpenMP traditionally targets loop-based parallelism and so we believe that our approach is more general and targets problems with a higher granularity. Furthermore, OpenMP relies on a *fork-join* model of execution, while our approach relies on data-flow techniques and producer-consumer

synchronization for the scheduling of threads, which allows it to represent a wider-set of problems and provide better performance for irregular code.

The introduction of the OpenMP 3.0 specification tried to address these shortcomings by extending the standard with the concept of tasks to accommodate irregular applications. This increased the class of programs that can be handled by OpenMP, however, it still doesn't allow the explicit specification of general dependencies amongst tasks, which our model naturally does. Recent efforts [16] within the OpenMP community have been made in this direction, however, it is not yet part of the formal specification. We find these efforts as an indication of the growing recognition of the community towards the benefits of data-flow based programming and scheduling techniques.

OpenMP Compiler for the Cell

The IBM Research Compiler targeting the Cell architecture [39] ports the OpenMP standard to the Cell processor. It manages the execution and synchronization of the parallelized code and handles data transfers via a compiler-controlled software cache. In addition to the aforementioned differences between our approach and OpenMP in general, our approach on the Cell relies on data-flow techniques and data-flow caching policies to schedule threads and prefetch & manage their data. Moreover, because our proposed approach relies for compilation on the available Cell platform compilers it can benefit from the latest optimizations and vectorization techniques provided by this compiler or any other to optimize the code of the DDM threads that will run on the Cell cores.

2.3.6 Cilk

Cilk [21] is a parallel programming extension to the C language that adds a few keywords for facilitating parallelism. The keywords are mainly used to *spawn* functions as asynchronous parallel tasks and synchronize amongst the tasks using a barrier-like *join* method. When removing the Cilk-specific keywords from a program the result is still a valid sequential C program. Cilk programs are preprocessed to C and then compiled and linked to a runtime library.

The programmer is responsible for identifying the task functions that can run in parallel and is also responsible for synchronizing amongst the tasks. The run-time manages the scheduling of the tasks using a *work-stealing* scheduling policy. At runtime, Cilk programs can be viewed as directed acyclic graph (DAG) that unfolds dynamically as the program executes. Cilk employs a special consistency model that is called *DAG consistency* [20], which is a relaxed consistency model defined on the DAG of the tasks that make the parallel computation. Under this model, a *read* can see a *write* only if there is some serial execution order consistent with the dependencies of the DAG (a read always respects the dependencies in the DAG). Thus, the writes performed by a task are seen by its successors, but tasks that are incomparable in the DAG may or may not see each other's writes. In this thesis we employ a similar concept for maintaining the memory consistency.

The *fork-join* approach adopted by Cilk is very well-suited for expressing recursive algorithms (e.g., divide-and-conquer), however, unlike our approach Cilk does not rely on data dependencies for the scheduling of tasks (neither at compile-time nor run-time), which misses part of the potential parallelism in many programs. Indeed, it has been shown that Cilk cannot efficiently schedule workloads in dense linear algebraic algorithms [62].

2.3.7 Intel Threading Building Blocks (TBB)

Intel Threading Building Blocks (TBB) [95] is a C++ template library developed by Intel to facilitate parallel programming. It abstracts the low-level parallel resources of the machine and the threading mechanism by providing a set of data structures and algorithmic skeletons that supports the execution of tasks. It also provides a set of concurrent containers (queues, vectors, hashmaps, etc) and synchronization constructs (mutex constructs and atomic operations).

Similar to Cilk, the TBB runtime implements a tasks-stealing scheduling policy and adopts a *fork-join* approach for the creation and management of tasks. Consequently, it suffers the same shortcomings as Cilk.

2.3.8 Streaming Platforms

RapidMind [56] is a programming model that provides a set of APIs, macros and specialized data types to write streaming-like programs that targets general multi-cores and advanced GPUs and the Cell. It was recently integrated into Intel's data parallelism platforms. Cell-Space [87] is a framework for developing streaming applications on the Cell using a high-level coordination language out of components in a component library. It provides a runtime that handles scheduling, data transfers and load-balancing. Compared to the streaming approaches, we place our approach as a more general one, as it doesn't require the use of any streaming abstraction and can be used for a wider range of applications.

2.3.9 Software Caches on the Cell

In this section we review a number of the software caches that have been proposed for managing the local store memories of the SPE cores on the Cell processor. Please refer to Section 2.2.1.9 for an overview of the Cell processor.

A hierarchical, hybrid software-cache architecture to manage and optimize data transfers on the Cell has been proposed in [45]. At compile time, the memory accesses patterns are classified into two types: high-locality and irregular. Memory references are then directed into using cache structures optimized according to each types. This cache architecture allows the compiler to enable high-level loop optimizations and transformations to improve the performance and reduce the overhead of the software cache. In [26] direct buffering and software cache techniques are integrated to manage data transfers on the Cell efficiently using both techniques in the same program. It uses compile-time analysis and runtime time maintenance to achieve their goal.

The Multidimensional Software Cache (MDSC) is a software cache for scratch-pad based system that targets the Cell processor. The MDSC is optimized for applications that has a working-set that doesn't fit entirely in the cache and ones that access multi-dimensional data structures. It does so by storing 1- to 4-dimensional blocks in the cache and indexing the cache blocks by the matrix indices rather than using linear memory addresses. This minimizes memory transfer time (as it groups memory requests) and the number of cache access since it exploits the multidimensional access behaviour of the program. The cache supports a fully-associative or a set-associative organization and a FIFO or round-robin replacement policy. The evaluation of the MDSC doesn't account for parallel execution and doesn't address the problem of coherence. To utilize MDSC, data accesses in the program has to be manually replaced by special API calls.

Along with the software cache in [39], all of the aforementioned software caches perform cache directory operations on the SPE, in contrast with our proposed software cache, in which these operations are performed on the PPE and overlapped with the execution of code on the SPEs to hide the operations overheads. Moreover, our software cache enables data re-use and maintains coherency utilizing a mechanism that avoids expensive update/invalidate operations by using a simple Directed Acyclic Graph (DAG) consistency model. Most notably, our proposed software

cache is utilized at the scheduling and data management levels and contains elements specific to DDM.

Lee et al. [77] developed a software cache for the Cell processor, which supports a coherent globally shared memory view at the page-level utilizing a centralized lazy release consistency. The programmer uses a set of macros and runtime calls that support an SPMD-style programming model. Each read/write access in the SPE code is replaced by a runtime call to check whether the data is available in the local store, and to automatically fetch it if it's not. Similar to our approach the tasks of synchronization and coherence are managed by the PPE. An evolution of this work [76] extends the proposed technique to implement a software shared virtual memory (SVM) system for heterogeneous multi-core accelerator clusters with explicitly managed memory hierarchies that are connected with an interconnection network. The target cluster consists of a single manager node and many compute nodes. A coherence and consistency protocol, called hierarchical centralized release consistency (HCRC) is proposed to provide a shared memory software SVM system across the system.

2.3.10 Self-Distributing Virtual Machine (SDVM)

The Self-Distributing Virtual Machine (SDVM) is an adaptive, self-configuring and self-distributing virtual machine that manages the execution of tasks on a clusters of heterogeneous nodes. It adopts a decentralized peer-to-peer structure and emphasizes the ability to handle different configurations, systems and resource types. It also automatically adapts to changes in the system resources. The SDVM also supports a self-managed scheduling approach and supports the migration of code and data within the cluster.

Programs are composed of a number of *micro-threads*. Each *micro-thread* consists of a number of instruction and is associated with one or more *micro-frames*. The *micro-frames* contain the

micro-thread data and pointers to the *micro-frames* that require results generated by the *micro-thread*. Once all the argument data in a *micro-frame* is ready the associated *micro-thread* can execute.

The SDVM comprises two layers: execution and communication. Both are composed of a number of modules or managers. The execution layer manages the execution of the *micro-threads* after their associated *micro-frames* are ready and when the required resources are available. It also passes the resulting arguments destined for the consumer *micro-frames*.

The communication layer manages the exchange of messages between the nodes in the cluster. Most importantly, it handles messages related to the exchange of data objects amongst the nodes. Each data object in the system is associated with a home-site directory that keeps track of the physical location of the data object as it is moved and/or replicated. When requesting access to a non-local data object, its directory is queried to find the current location and a request is forwarded to that location. When writing to one copy of a data object, the home site is informed so as to update all the copies with the new value.

The SDVM supports the addition and removal of nodes from the cluster at any time. It also supports migration of code and data between the nodes and utilizes a check-point mechanism to recover from failures.

The SDVM shares a number of similar concepts with our work, most importantly it adopts a data-availability approach for the scheduling of threads and supports execution on a cluster of nodes. However, the current design of the SDVM doesn't target multi-core nodes. Furthermore, it is more of a middle-ware that targets loosely coupled machines forming a cluster, which makes it more suitable to the domains of large-clusters and grid computing due to the high-overhead involved during the execution.

2.4 Data-Driven Multithreading

The Data-Driven Multithreading (DDM) [73] is a non-blocking multi-threading model that combines the benefits of the Data-Flow model in exploiting concurrency with the highly efficient sequential processing of the commodity microprocessors. The core of the DDM model is the Thread Synchronization Unit (TSU), which is responsible for the scheduling of threads at run-time based on data availability. Scheduling based on data availability can effectively tolerate synchronization and communication latencies.

A DDM program consists of several threads of instructions. The threads have Producer-Consumer relationships and are grouped into DDM Blocks. A DDM block is equivalent to a loop or a function in high-level languages. The TSU schedules a thread to run only after all the producers of this thread have completed execution, which ensures that all the data this thread needs is available. Once the execution of a thread starts, instructions within a thread are fetched by the CPU sequentially in control-flow order, thus exploiting any optimization available by the CPU hardware.

The threads are identified by the *ThreadId* and *Context*. The *context* uniquely identifies the dynamic invocations of each thread. At compile-time a program is partitioned into the *synchronization graph* and the code of the threads. Every node in the graph represent one thread along with the associated template.

The synchronization template of the thread specifies the following attributes:

- **The Instruction Frame Pointer (IPF):** points to the address of the first instruction of the thread.
- **The ReadyCount (RC):** a value equal to the number of producer-threads this thread needs to wait until starting to execute.

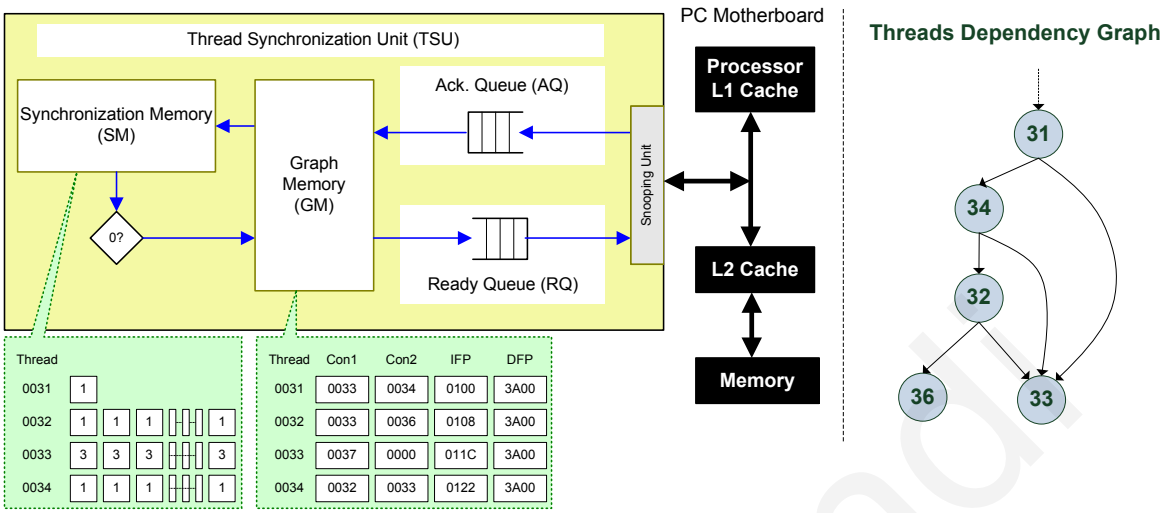


Figure 7: DDM Node

- **The Data Frame Pointer List (DFPL):** a list of pointers to the data inputs/outputs assigned for the thread.

- **The Consumer List (CL):** a list of this thread's consumers that is used to determine which *ReadyCount* values to decrement after the thread completes its execution.

Figure 7 illustrates a DDM node. The Graph Memory (GM) in the TSU holds the synchronization templates of all the threads. The Synchronization Memory (SM) holds the Ready Count values for each thread invocation. The TSU communicates with the processor via two queues. The processor reads the address of the next thread to execute from the Ready Queue (RQ) and stores the information of completed threads in the Acknowledgement Queue (AQ). The TSU fetches the completed threads from the AQ, finds their consumers in the GM and then updates the Ready Count of the consumers in the SM. If the Ready Count of any consumer becomes zero, it it is deemed executable and so is inserted in the RQ where it awaits for execution.

DDM can improve the locality by implementing deterministic data prefetching using data-driven caching policies called CacheFlow [72]. CacheFlow policies include firing a thread for execution only if the code and data of the thread are present in the cache. Furthermore, blocks

associated with threads scheduled to execute in the near future are not replaced until the thread finishes its execution. Results of applying CacheFlow have shown that CacheFlow reduces cache misses considerably, even on caches of small sizes [72].

The DDM model had two implementations: the Data-Driven Network of Workstations (D^2NOW) [73] and *Thread Flux* TFlux [113]. The work presented in this thesis is the third implementation of DDM.

2.4.1 The Data-Driven Network of Workstations

The first implementation of DDM was the Data-Driven Network of workstations (D^2NOW) [73]. D^2NOW was a simulated cluster of distributed machines augmented with a hardware Thread Scheduling Unit (TSU), which exhibited tolerance to long memory and communication latencies. The TSU was attached to the COAST (Cache On A STick) L2 Cache slot of Pentium workstations and thus it had an implicit snooping interface to the Pentium microprocessor. Figure 8 illustrates the architecture of the D^2NOW.

The Thread Scheduling Unit (TSU)

The TSU (shown in Figure 9) comprises three units: the Thread Issue Unit (TIU), the Post Processing Unit (PPU) and the Network Interface Unit (NIU). The PPU is responsible for updating the Ready Count of the consumer threads. The TIU is responsible for scheduling threads deemed executable by the PPU. The NIU is responsible for the communication between the TSU and the interconnection network. The Network Interface Unit (NIU) consists of the Transmit Unit and the Receive Unit.

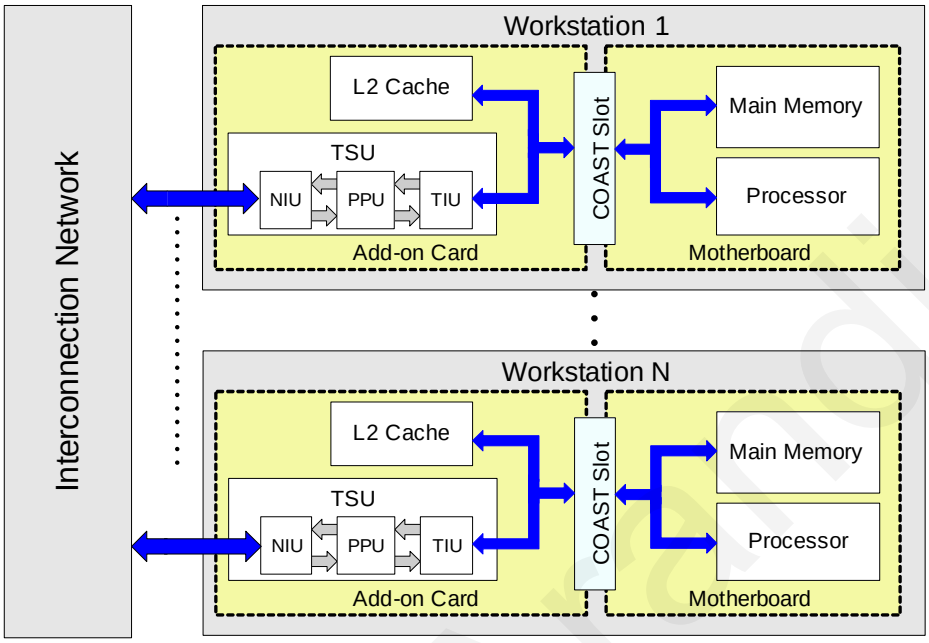


Figure 8: The D^{2}NOW Architecture (Figure from[71])

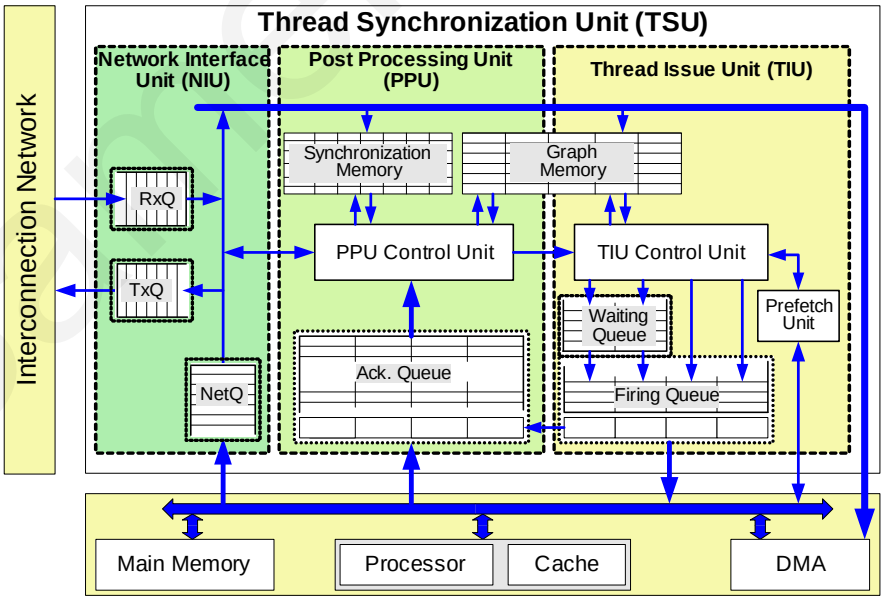


Figure 9: The TSU internal structure (Figure from [71])

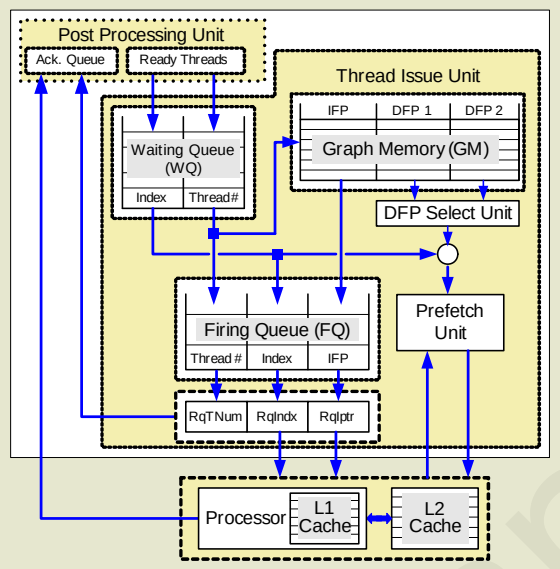


Figure 10: The TSU with the basic prefetch CacheFlow policy (Figure from [71])

CacheFlow

D^{2}Now exploited short-term optimal cache placement and replacement policies to further improve the performance. Three implementations of the CacheFlow policy were examined.

- The Basic Prefetch CacheFlow: the data of the threads scheduled for execution in the near future is perfetched into the cache. Once the prefetching for a thread finishes, it is placed in the Firing Queue (FQ) and where it awaits for its turn to be executed. Figure 10 illustrates the hardware of the basic CacheFlow policy

- CacheFlow with Conflict Avoidance: the prefetched data belonging to threads waiting in the FQ is protected from eviction until the corresponding threads are executed.

- CacheFlow with Thread Reordering: the sequence of executable threads is re-ordered before they are inserted in the FQ to takes advantage of locality.

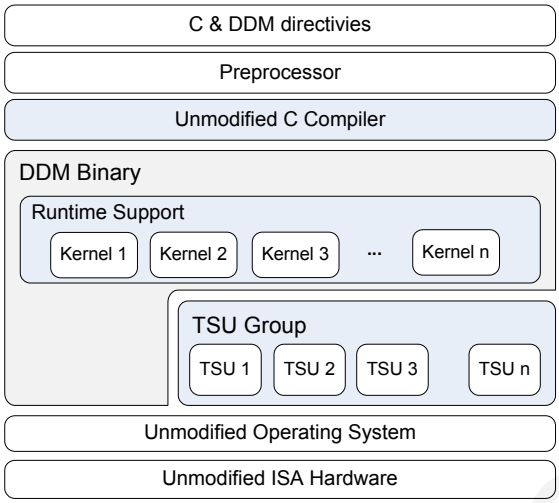


Figure 11: The TFlux Platform (Figure from [113])

The Evaluation of the D^{2}NOW showed that overall, it proved to be a promising architecture as it effectively hides synchronization and communication latency. In addition, the CacheFlow policy reduces significantly the cache miss ratio, resulting in close-to-linear speedups.

2.4.2 Thread Flux

Thread Flux (TFlux) [113, 112] is a platform that supports the DDM model of execution independently of the underlying architecture. It provides a runtime support that is built on top of a commodity operating system. TFlux is composed of a collection of entities in a layered design that abstracts the details of the underlying machine. Figure 11 depicts the layered design of the TFlux system. The most important components are the Runtime Support and the TSU Group. The Runtime Support runs on top of an unmodified Unix-based Operating System and shields the details of the particular implementation of the TSU. The functionality of the runtime is supported by simple user-level processes called the *Kernels*, which manage the execution of the thread and the communication with the TSU.

The TSU Group is a single unit that is responsible for the scheduling of threads based on data availability. It comprises a global part common to all the cores in the system and TSU units that serve each core individually.

TFlux Toolchain

TFlux provides a preprocessor supporting a set of compiler directives that facilitate developing DDM programs. Applications are easily ported to TFlux by augmenting C code with the directives. The preprocessor automatically generates TFlux code, which includes all the necessary calls to the TFlux runtime system. The code can be compiled using any commodity C compiler, thus producing binaries for any ISA.

TFlux Implementations

TFlux has 3 implementations:

- TFluxHard

  TFluxHard is a shared memory Chip Multiprocessor augmented with a hardware implementation of the *TSU Group*. The TSU Group is attached to the system network as a memory-mapped device. The communication between the TSU and the CPU is performed via the Memory-Mapped Interface (MMI). The Grouping of multiple TSUs (each servicing one core) into one single unit aims at decreasing the additional interconnection cost. Figure 12 presents a TFluxHard chip configured with 4 cores. TFluxHard was evaluated using a simulated machine, which was built using the Simics full-system simulator [78].

- TFluxSoft

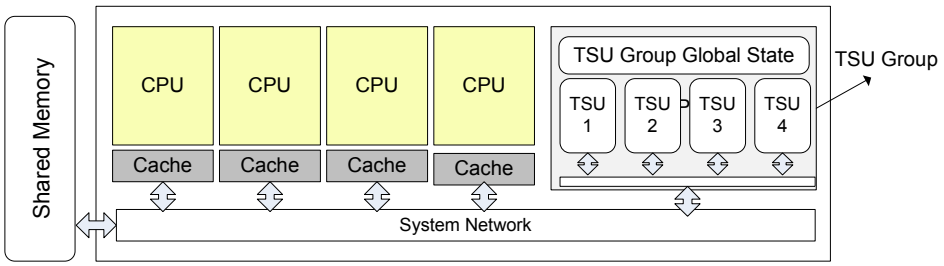


Figure 12: TFluxHard chip with 4 cores (Figure from [113])

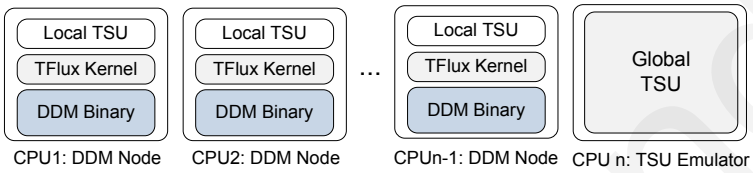


Figure 13: TFluxSoft system executing on a system with n CPUs (Figure from [113])

This implementation primarily targets commodity multi-core processors with a single shared address space and a hardware cache coherency. The TSU is implemented as a software module that executes on one of the cores of the processor. This module emulates the functionalities of the hardware TSU and so it is called the TSU Emulator. To avoid overloading the core that provides the TSU functionality, part of the operations is distributed to be executed by local TSUs or *kernels*. Figure 13 depicts the execution of the TFluxSoft system on a system with n CPUs.

- TFluxCell

  This implementation targets the Cell Processor and has the TSU Emulator running on the PPE core and the threads on the SPE cores. Part of the communication between the kernels running on the SPEs and the TSU on the PPE is implemented using DMA calls and the rest using the fine-grain mechanisms of *mailboxes* and *signals*. The following scheme was used for transferring the produced/consumed data amongst the threads. When a thread finishes execution, the produced data is exported to a shared buffer in main memory and before the

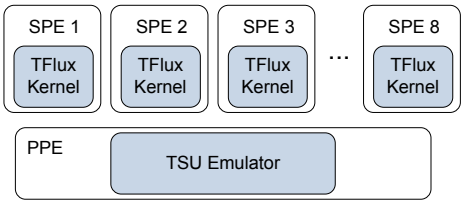


Figure 14: The TFluxCell system (Figure from [113])

consumer thread starts, this data is imported from the buffer into the Local Store (LS) of the SPE where the consumer thread will be executing. Figure 14 illustrates the TFluxCell system.

The evaluation of TFlux showed that the achieved speedup is close to linear. Most importantly, the speedup results are stable across the various platforms, thus indicating that TFlux allows exploiting the benefits of DDM on different commodity systems.

2.4.3 The Data-Driven Multithreading Virtual Machine

The work presented in this thesis is the third implementation of the Data-Driven Multithreading model, in which we advance its state-of-the art by designing, implementing and optimizing the Data-Driven Multithreading Virtual Machine (DDM-VM).

D^{2}NOW utilized a hardware TSU and supported distributed DDM execution on a cluster of *single* processor nodes, while DDM-VM supports distributed DDM execution on a cluster of *multi-core* nodes. D^{2}NOW utilized CacheFlow to optimize the performance, while DDM-VM uses the concept of CacheFlow to implement a prefetching software cache for the automatic management of software-managed memory hierarchies.

The software implementations of the TFlux platform and the DDM-VM utilize a software TSU, however, DDM-VM supports distributed DDM execution on clusters of multi-core nodes. It

further develops a prefetching software cache for software-managed memory hierarchies on heterogeneous multi-cores. DDM-VM also applies further performance optimizations and compares favorably with state-of-the-art similar systems. Finally, DDM-VM supports runtime dependency resolution, which expands the class of programs that can be mapped to the DDM model and has the potential to improve the programmability.

Chapter 3

The Data-Driven Multithreading Virtual Machine (DDM-VM)

3.1 Introduction

The Data-Driven Multithreading Virtual Machine (DDM-VM) is a parallel software platform that supports Data-Driven execution on conventional control-flow multi-core systems. The Data-Driven Multithreading model combines the latency tolerance and distributed concurrency mechanisms of the data-flow model with the efficient execution of the control-flow model. The DDM-VM utilizes DDM scheduling for exploiting the resources of multi-core architectures and tolerating synchronization and memory latencies. It employs data-flow concurrency for scheduling threads and efficient sequential execution of instructions within a thread. The scheduling of threads is orchestrated by the Thread Scheduling Unit (TSU), which is implemented as a software module running on one of the cores. The TSU is aided by the runtime that supports DDM execution on the rest of the cores. The DDM-VM targets homogeneous and heterogeneous multi-core architectures with software-managed memory hierarchies. A software prefetching cache based on data-driven caching policies is employed for handling the software-managed memory hierarchies. The DDM-VM supports DDM execution within a single multi-core node and across multiple multi-core nodes connected using an off-chip network.

In this chapter we present the architecture of the DDM-VM and describe its two implementations. In addition, we highlight the design and implementation the *Software CacheFlow* (S-CacheFlow) developed for managing the memory hierarchy in the DDM-VM$_c$.

This chapter describes the support for DDM execution within a multi-core processor or a *node*. Extending the DDM-VM design to support distributed DDM execution across multiple nodes is described in Chapter 4. Details of the programming methodology and tools are given in Chapter 5.

3.2 The Data-Driven Multithreading Virtual Machine (DDM-VM)

The Data-Driven Multithreading Virtual Machine (DDM-VM) is a virtual machine that supports DDM execution on homogeneous and heterogeneous multi-core systems. DDM-VM virtualizes the parallel resources of the underlying machine and uses a unified representation for DDM programs. The representation is flexible enough to incorporate additional information needed to optimize the DDM execution for different target architectures. The DDM-VM composed from the Thread Scheduling Unit (TSU) and the runtime system, handles the tasks of thread scheduling, execution instantiation and data management implicitly.

DDM-VM programs are mapped into the code of the DDM threads and the *synchronization graph* or *meta-data* of the threads, such as the *ReadyCount* (RC) and the consumer/producer dependency relationships of each threads. The virtual machine uses the *meta-data* to schedule threads based on data-availability; a thread is scheduled for execution when all its producers finish execution. The scheduling of threads is interleaved with their execution, thus shortening the critical path of the application. In the case of architectures with a software-managed memory hierarchy the DDM-VM prefetches the thread data from the main memory to the cache of the processor before starting its execution.

Thread Scheduling Unit (TSU)

The cornerstone of supporting DDM execution is the implementation of the Thread Scheduling Unit (TSU) responsible for scheduling threads dynamically based on data-availability. The DDM-VM implements the TSU as a software module running on one of the cores in the system, leaving the rest of the cores for threads execution. Previous implementations of DDM have realized the TSU as a hardware module attached on the COAST (Cache On A STick) interface [73], a simulated memory-mapped hardware module [113], and as a software module [113] similarly to this work. Naturally, the overheads of a software implementation (in terms of the cost of the TSU operations) is larger when compared to a hardware one. However, a software implementation allows obtaining the benefits of DDM execution on existing off-the-shelf systems without changing the underlying architecture or adding new hardware components. In addition, the overlapping of the TSU work with the execution of threads reduces such overheads, especially when increasing the granularities of the threads, as will be demonstrated in the Evaluation Chapter.

The DDM-VM architecture has two implementations: The Data-Driven Multithreading Virtual Machine for the Cell (DDM-VM$_c$) and the Data-Driven Multithreading Virtual Machine for Symmetric Multi-cores (DDM-VM$_s$). Figure 15 depicts both implementations. We describe both implementations in the following sections.

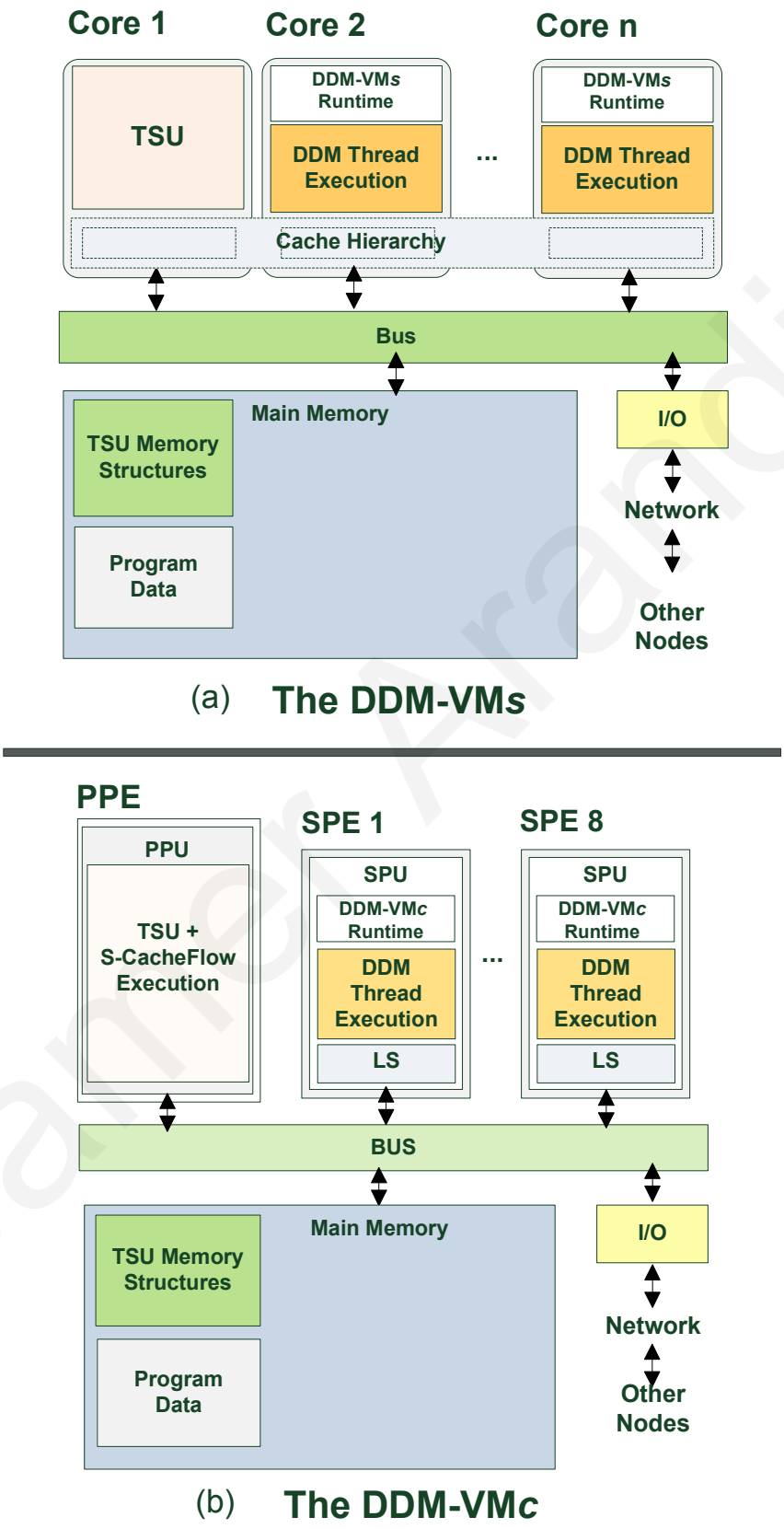


Figure 15: The two implementations of the DDM-VM architecture (a) DDM-VM$_s$ (b) DDM-VM$_c$

3.3 The Data-Driven Virtual Machine for the Cell (DDM-VM$_c$)

The Data-Driven Virtual Machine for the Cell (DDM-VM$_c$) is the DDM-VM implementation targeting heterogeneous multi-cores with a host/accelerator organization and a software-managed memory hierarchy. The Cell Broadband Engine processor [63] (Cell/B.E. or Cell for short) is the principal representative example of such architectures and thus has been chosen as the target for this implementation. For a detailed description of the architecture of the Cell processor please refer to 2.2.1.9.

The Thread Scheduling Unit (TSU) is implemented as a software module running on the PPE core. The execution of the program threads takes place on the SPE cores. The communication between the TSU and the executing threads is facilitated via DMA calls. A software prefetching cache module in the TSU, called Software CacheFlow (S-CacheFlow), manages data transfers and prefetching automatically. Thread scheduling and S-CacheFlow operations running on the PPE are interleaved with the execution of threads on the SPEs, thus shortening the critical path of the application. All these operations are implemented by the runtime requiring no intervention from the programmer. Figure 16 illustrates the architecture of the DDM-VM$_c$.

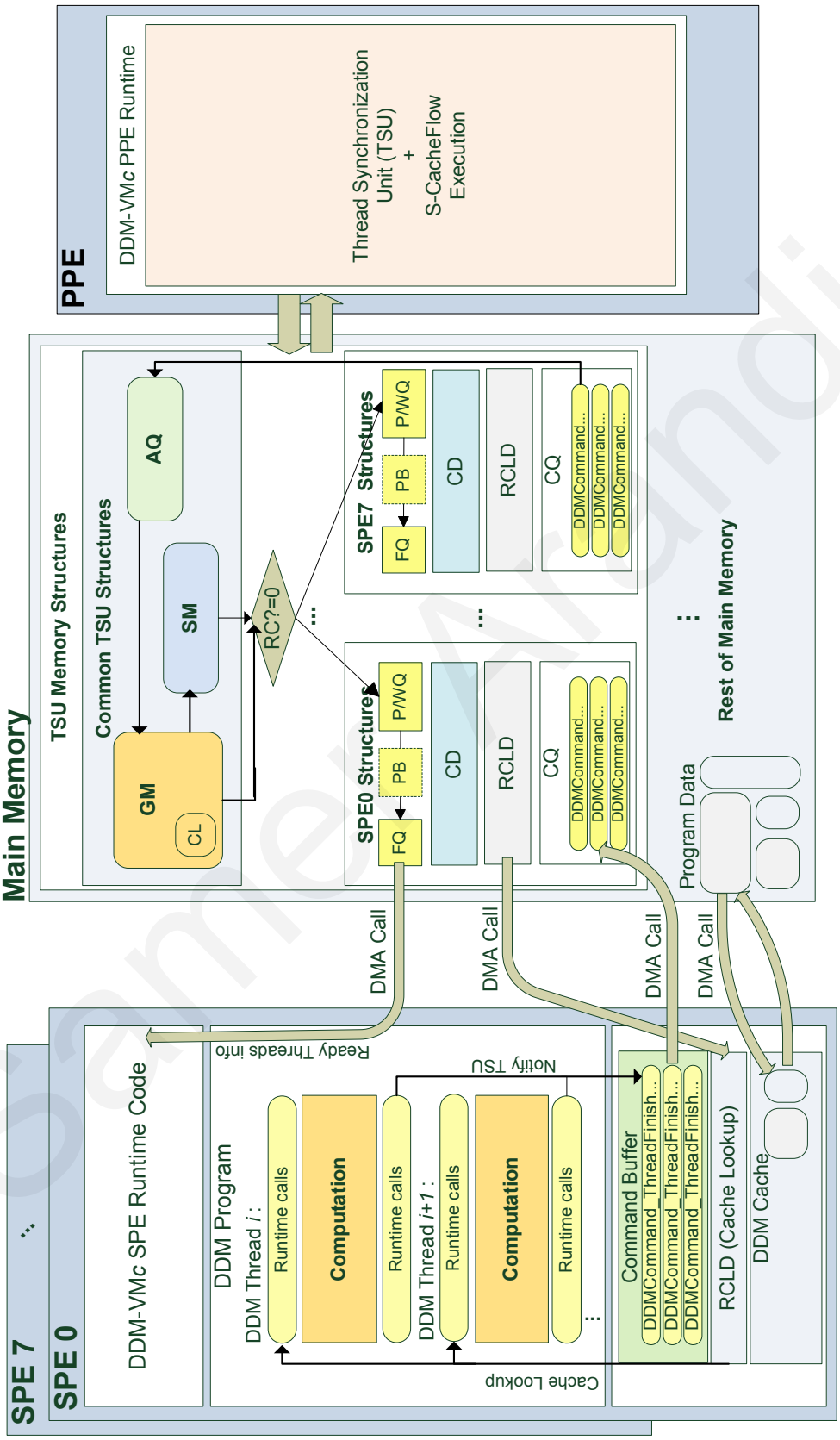


Figure 16: The Architecture of the DDM-VM$_c$

In the following subsections we present the motivation behind our work on the Cell and the rationale of the design, followed by a detailed description of the implementation of the Thread Scheduling Unit (TSU) and the TSU-Processor interface. The Software CacheFlow implementation is described in 3.4.

3.3.1 Motivation and Design Rationale

A close examination of the Cell processor design and the DDM model reveals a matching on many levels that motivated our work on the Cell.

Core Specialization

The specialization of the Cell cores between control and execution, matches the decoupling of synchronization and execution adopted by the DDM model. This led us to map the TSU to run on the general purpose PPE core, while leaving the SIMD SPE cores to execute the threads. This is an efficient utilization of the Cell heterogeneous resources, since the PPE is already used for controlling execution on the Cell, as it runs the Operating System, acts as a coordinator for the other cores (SPEs), and provides them with various services. One can look at the TSU services as an additional kernel service. Another point to consider here is that the PPE -as a general purpose processor- is well-suited for executing the code of the TSU that heavily uses branches and control-flow structures, while the threads are well-suited to run on the SIMD SPE cores, which are optimized for executing computational loads.

Software-managed Memory Hierarchy

Another motivating factor was the software-managed memory hierarchy, which is challenging to manage but offers a great opportunity for improving performance. This organization allows

deterministic execution at the SPEs once the required data resides in the LS memory, as no cache misses is encountered. Therefore, optimal performance is achieved by increasing the time the SPEs are working on data in their *near* LS and diminishing the time spent waiting for data to come from the *far* main memory. This is usually achieved using techniques like *double buffering*, in which the programmer splits the program data into multiple sets and work on one set while asynchronously fetching the next one from main memory. This interleaving between computation and data communication is the principal way the Cell tries to avoid the Memory Wall. Nevertheless, a considerable effort from the programmer is required to structure the code to take advantage of this technique. If the data-driven prefetching policies employed with DDM is applied to the local store, double-buffering will be achieved automatically with no programmer intervention. In fact, this capability of controlling the LS memory by software allows DDM more control, compared to conventional hardware caches, to get the maximum potential of its pre-fetching techniques. More details on the management of the memory hierarchy of the Cell is presented in Section 3.4.

Context-switching Overheads

Finally, because each SPE core supports a single program context at any time, conventional context-switching at the level of the O.S. is very expensive [5]. Each SPE has 128 by 128-bit register file, DMA command queue status, and 256 KB LS memory that ought to be (at least partially) saved. Therefore, a self-managed approach is advised for handling concurrent execution within the same application. The DDM model provides this by default, since the scheduling of DDM threads is done internally by the TSU incurring no context-switching and appearing to the O.S as a run-to-completion type of execution.

After presenting the motivation behind our work, we proceed to describe the implementation details of the DDM-VM$_c$.

3.3.2 The Thread Scheduling Unit (TSU)

The Thread Scheduling Unit (TSU) is the core of the DDM model. It holds the *meta-data* of the threads and use it to schedule the threads dynamically at runtime based on data-availability. It also manages the prefetching of each thread data using the Software CacheFlow module. In this section we present the implementation details of the TSU. We describe the memory structures holding the TSU state and the TSU operations.

The TSU Memory Structures

As the TSU runs on the PPE, the structures holding the thread *meta-data* and the state of the TSU are allocated in main memory. Part of the structures are common for all the SPEs and the rest are allocated per SPE. The common TSU structures include:

The Graph Memory (GM): holds the *synchronization template* of each thread. This includes: the thread identifier (ThreadId), Instruction Frame Pointer (IFP), Consumers List, number of Data Frame Pointers, the Ready Count (RC) value and the thread attributes. The attributes include: the scheduling policy to use for the thread along with a scheduling value, the SM implementation to use (3 different implementations are available: *direct*, *associative* and *hybrid*), a mask value used when the *direct* implementation is selected, and the *arity* of the thread specifying the loop nesting level for threads implementing loops. The final part of the template is held in the Consumer List (CL) auxiliary table. Note that the template holds the number of DFPs, the values of the DFPs (the addresses of the input/output data), however, are retrieved at runtime by calling a helper-function.

The Consumer List (CL): The GM entry contains two fields, *Con1* and *Con2*, which can hold the thread identifier for one or two consumers. If the thread has more than two consumers, the CL holds the list of consumer threads for that thread. In that case, *Con1* is set to zero and *Con2* is set to point to the entry of the first consumer in the CL. Each consumer entry points to the *next*

one and the entry of the last consumer has the value of *next* set to -1. This is an optimization as previous DDM work has shown that most threads have two consumers [73].

The Synchronization Memory (SM): holds the RC values for each invocation of a DDM thread. The SM entries are uniquely indexed using the *context* of the invocations. The RC value in the GM entry is used to initialize the RC entries in the Synchronization Memory. As the performance of the SM is critical to the overall system performance, we have utilized three different implementations of the SM. Details of the implementations are presented in Section 5.6 and their performance evaluation is presented in Section & 7.2.2.2.

The Acknowledgement Queue (AQ): holds requests to decrement the RC of one or more invocations of consumer threads. The requests are enqueued when a producer thread finishes execution. The request include the consumer identifier, *context*, the operation flags, two values used when updating multiple invocations and the updated value by which the consumer(s) RC is decremented (set to 1 by default).

The per-SPE TSU structures include:

The Command Queue (CQ): holds the DDM Commands sent by the executing threads. The commands inform the TSU that a thread has finished execution and indicate the consumer thread(s) invocation(s) to decrement their RC. The entries hold information similar to the ones in the AQ entries. However, the *consumer_id* field indicates the order of the consumer thread in the CL (the first, second, third consumer, etc.) as opposed to the *consumer_id* field in the AQ which holds the ThreadId of the consumer. Moreover, the *OP* field carries an extra flag which indicates that the currently executing thread has finished execution.

The Waiting Queue (WQ): holds the information of threads which RC reached zero and are waiting for prefetching to start. This includes the ThreadId and *context*.

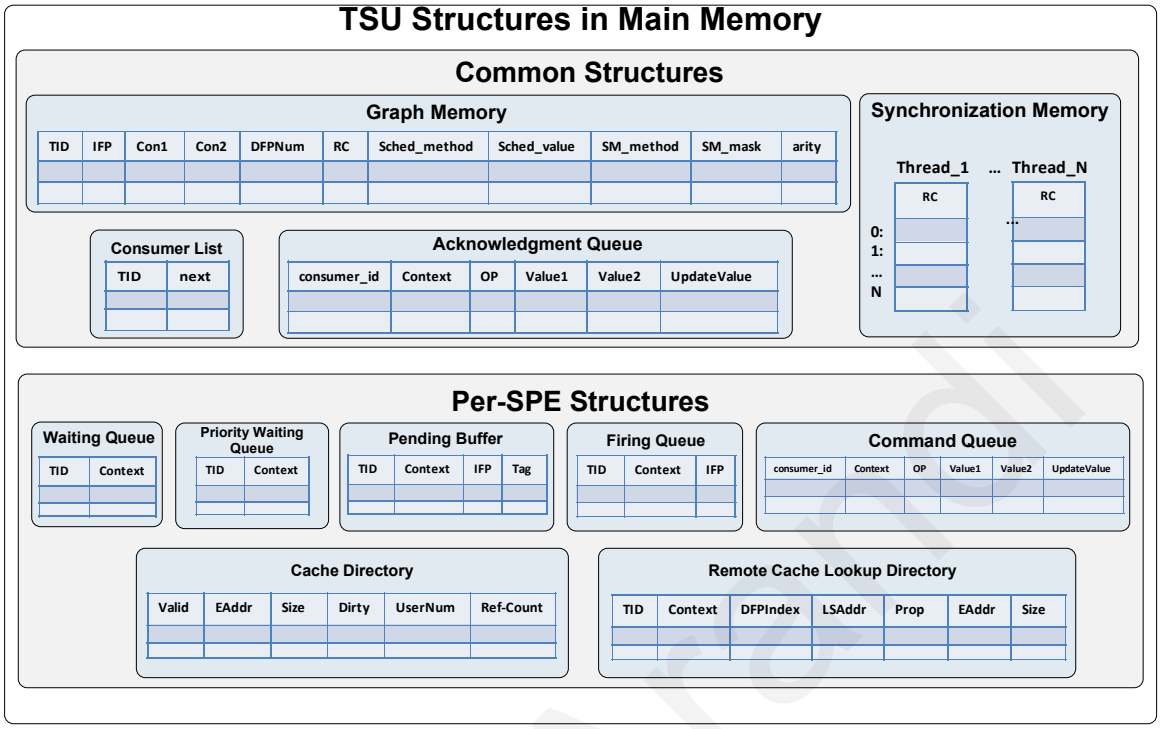


Figure 17: The TSU Structures in Main Memory

The Priority Waiting Queue (PWQ): this queue is identical to the WQ, however, its entries have a higher-priority. It holds the information of threads that were dequeued from the WQ but their prefetching was not started due to unavailable space in the LS.

The Pending Buffer (PB): holds information of threads whose prefetching is started (by issuing DMA transfers) and are waiting for its completion. Each entry records the information of the thread along with a unique 5-bit *tag* used for checking the completion of the DMA transfers. In the distributed configuration of S-CacheFlow (described in 3.4), this buffer is moved to the LS.

The Firing Queue (FQ): holds the information of threads whose data has been prefetched into the LS and are ready to execute. This includes the ThreadId, IFP and the *context*. In the distributed configuration of S-CacheFlow (described in 3.4). This queue is moved to the LS.

The structures required for the operation of the S-CacheFlow are allocated in main memory as well. This includes the Cache Directory (CD) and the Remote Cache Lookup Directory (RCLD),

which are allocated per-SPE. Detailed description of these structures is presented in 3.4. Figure 17 depicts the TSU structures in main memory.

The LS memory of the SPEs (shown at Figure 16) holds (i) the code of the DDM threads linked with the VM runtime library and (ii) the rest of the S-CacheFlow structures including the part of the LS which holds the data of the DDM threads. We refer to this LS part as the *DDM Cache*.

3.3.2.1 Thread Execution and TSU Operations

Thread Execution

The DDM thread execution takes place on the SPEs and consists of two types of operations or phases: computation and synchronization. The synchronization operations are performed by the runtime on the SPE, which communicates with the TSU via the TSU-Processor interface. In particular, the runtime sends simple messages or DDM commands (will be defined shortly) to the corresponding TSU Command Queue (CQ) in main memory. Moreover, when a thread finishes execution, the runtime fetches the information of the next thread to execute from the corresponding FQ in main memory. A detailed description of the TSU-Processor interface is presented in the following section.

TSU Operations

The TSU running on the PPE core performs 3 main activities continuously until the termination of the program:

- Executes the commands in the CQs
- Decrement the RC of consumer threads

- Execute the S-CacheFlow

The activities and the TSU structures affect by each one are illustrated in Figure 18.

The TSU processes the commands in the CQs of all the SPEs in a round-robin fashion. The commands inform the TSU that the current executing thread on that SPE has finished. The commands also relay the information of which consumer thread(s) invocation(s) to decrement their RC. This information is enqueued as requests in the AQ.

The TSU processes the AQ requests to decrement the RCs and if any RC reaches zero, the corresponding thread invocation is scheduled for execution on the SPE core that is selected by the scheduling policy. This is done by inserting the thread information into the Waiting Queue (WQ) pertaining to the selected SPE. If the WQ is full the RC update operation is undone and no entry is removed from the AQ. Moreover, if a certain threshold of consecutive WQ insertions is reached, the TSU stops processing the AQ entries and starts processing the WQ entries. This priorities the prefetching of thread data and consequently ensures that more threads are ready for execution. We have set the threshold value empirically and permit controlling it using a configuration file.

Threads in the WQ are then processed by the S-CacheFlow module, which transfers the data each thread requires to the LS of the SPE along with the information needed to access this data, like the LS address where the data is transferred and its size. Once the transfers complete the thread is deemed ready to execute and its information is moved into the Fire Queue (FQ). Should the FQ becomes full, the execution of the S-CacheFlow module is not invoked.

3.3.3 The TSU-Processor Interface

The TSU-Processor interface specifies the communication mechanism to use between the core executing the TSU and the ones executing the threads, i.e., between the PPE and SPEs. This

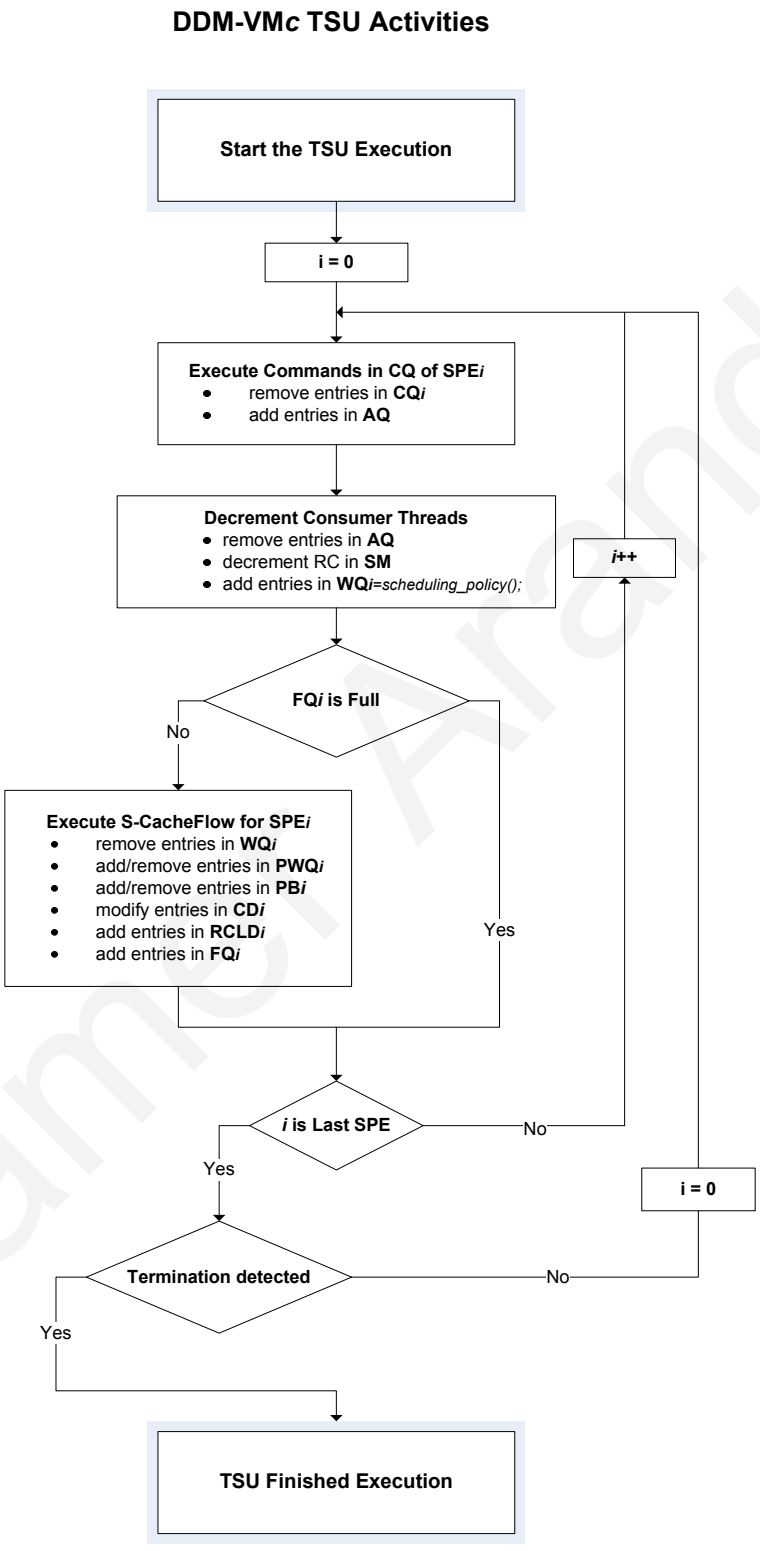


Figure 18: DDM-VM$_c$ TSU Activities

communication is abstracted by the virtual machine and implemented using the Cell low-level communication infrastructure. This interface performs two main tasks:

- Informing the TSU that the currently executing thread has finished execution and the consumer thread(s) invocation(s) to decrement their RC
- Providing the execution cores with the information of the next ready thread to execute

Next we discuss the Cell low-level communication mechanisms utilized by the interface and present the interface implementation details.

Low-level Communication Mechanisms

The Cell provides small granularity communication mechanisms (1 to 4 32-bit messages) via *mailboxes* and *signals* and coarse ones (128 byte to 32 MB per message) via DMA calls. We have opted for the latter as the basic communication mechanism in the interface for two main reasons: First, the size of the information exchanged when performing the main communication tasks is multiple times larger than 4-bytes. Second, when checking for the arrival of messages or data, it is more efficient for the PPE to poll a flag in main memory (set or cleared by DMA) than a flag in a *mailbox* or *signal* [5] (as the latter is an I/O operation).

TSU-Processor Interface - First Task

The first TSU-Processor interface task is implemented by sending simple messages or *DDM commands* to the TSU. The commands are first stored in a local Command Buffer in the LS of the SPE and then the buffer is copied into the Command Queue (CQ) associated with this SPE in main memory via DMA calls. We have utilizes an efficient circular CQ that requires minimal synchronization operations between the SPEs and the PPE. Instead of continuously issuing DMA

calls to read the value of the queue *head* pointer in main memory by the runtime on the SPE so as to calculate the correct size of the queue, we only do this when the runtime finds the queue nearly full. At that point, an asynchronous DMA call is issued to read these values from main memory. This call is overlapped with threads execution, thus greatly reducing its cost. Using this scheme the SPEs get a slightly *old* perception of the size of the CQ in main memory, however, continuous DMA transfers are avoided.

TSU-Processor Interface - Second Task

The second task consists of retrieving the next ready thread information. This includes the ThreadId, IFP and *context* of the thread, in addition to the S-CacheFlow information related to the thread data in the LS. This task is performed via DMA calls that fetch this information from the FQ and the S-CacheFlow structures in main memory into the LS.

The runtime takes advantage of the ability to issue asynchronous DMA calls to overlap sending of commands and data to main memory with thread execution. This well-known technique is called *multi-buffering*. We describe how we implemented it in Section 3.4.6. Referring back to Figure 16, the thick arrows represent the data movement and the various information exchanges taking place through the TSU-Processor interface in the DDM-VM$_c$.

3.3.4 The Scheduling Policy

So far we have discussed *how* and *when* a thread is deemed executable and the subsequent events occurring before and after execution. However, another important aspect of the process of scheduling is *where* to allocate or map the threads, which can have a considerable effect on performance in data-flow systems [43].

In the presence of enough information on the parallel program characteristics (e.g. task execution time, task communication and task dependencies) scheduling can be performed statically at compile-time [70]. In the absence of such information, however, it is performed dynamically at runtime [51, 19]. Scheduling techniques also vary in terms of locality-awareness, handling of task dependencies and the involved overheads but the common goal is -in general- reducing the overall execution time of the program, commonly called the *makespan*.

Supported Scheduling Policies

The DDM-VM$_c$ implements a number of scheduling policies that permit the programmer/compiler to control the mapping of threads to the SPE cores. The default policy distributes the threads invocations among the SPEs in a way that maximizes load-balancing. We denote this policy as the *dynamic* scheduling policy. The other implemented policies include the *static*, *round-robin*, and *modulo* policies. The *static* policy distributes the invocations of a specific thread to a specific SPE. The *round-robin* policy distributes the invocations of threads across the SPEs in a round-robin fashion. The *modulo* policy uses the *context*, uniquely distinguishing each thread invocation *modulo* the number of SPE cores to select the target SPE. This is a commonly used scheduling technique in data-flow systems that was -according to Gaudiot [43]- initially used in [46] .

The scheduling policies are assigned per-thread allowing for maximum flexibility. The DDM-VM$_c$ also supports using a *custom* policy, which gives the programmer/compiler the flexibility to implement a scheduling policy based on data locality or the dependency graph of the program or any other criteria.

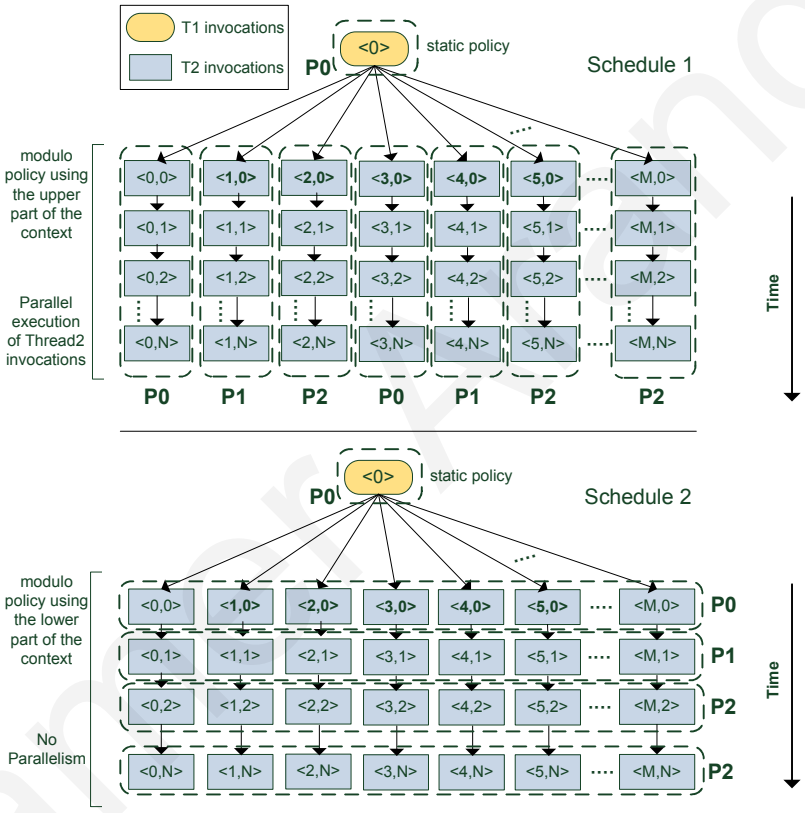


Figure 19: Two schedules using the *static* and *modulo* policies differently with a drastic effects on parallelism

Scheduling Example - Static and Modulo Policies

Figure 19 demonstrates two schedules each using the *static* and *modulo* policies to map the invocations of two DDM threads, representing two nested loops, to 3 SPEs. We label the dynamic invocations of the two threads with the value of the <*context*> distinguishing each invocation. As each invocation of *Thread 2* corresponds to one iteration of the inner loop in the original program, its *context* is depicted as <i,j> where *i* is the loop index of the outer loop and *j* is the loop index of the inner loop. The arrows represent the dependencies amongst the threads invocations.

In the first schedule, *Thread 1* is assigned a *static* policy that maps its invocations (it has one invocation) to SPE0 (abbreviated as P0). *Thread 2* is assigned a *modulo* scheduling policy that uses the upper part of *context* (the outer loop index) to distributed the invocations in a modulo fashion on the 3 SPEs. This is equivalent to distributing the corresponding outer loop iterations in a cyclic manner across the SPEs. In the second schedule the *modulo* policy uses the lower part of the *context* (the inner loop index), instead of the upper, for the distribution of the invocations.

It is clear that the first schedule enables parallel execution while the second one hampers it, which underlines the importance of selecting an appropriate scheduling policy.

Scheduling Example - Dynamic and RoundRobin Policies

Figure 20 demonstrates the *roundrobin* and *dynamic* scheduling policies used for an arbitrary program with multiple different threads. As time is relevant in the case of these two policies, the length of the threads rectangles represent their execution duration. The vertical distance between the dependency arrows start and end represent the time needed for performing the following tasks: informing the TSU that a thread has finished, decrementing the RC of its consumers, fetching the data of ready threads and eventually inserting the thread in the FQ. For simplicity we assume that

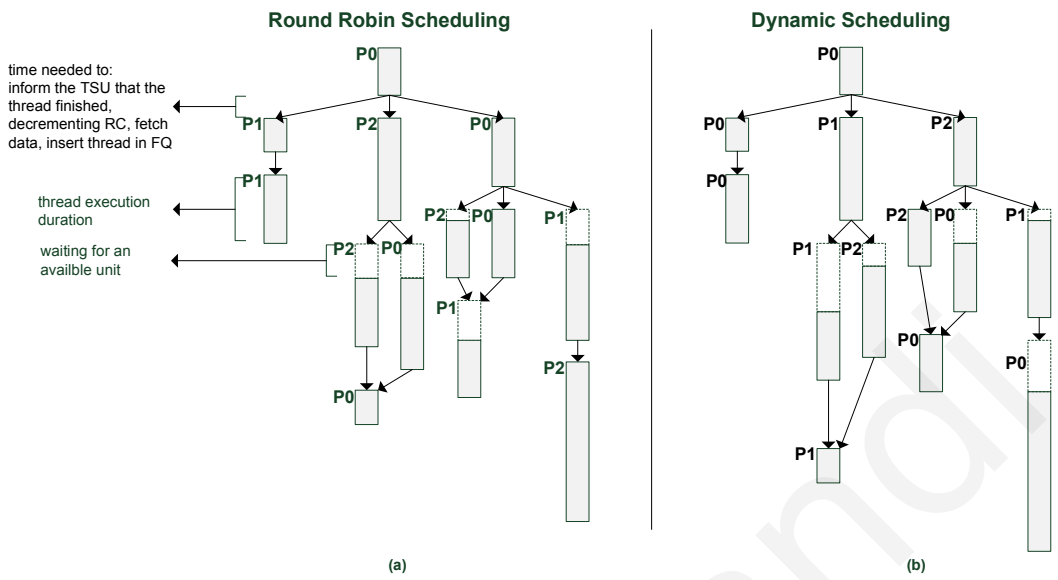


Figure 20: RoundRobin and Dynamic Scheduling Policies

these steps are accomplished in a constant time and that the scheduling decision is made mid-way. The empty dashed rectangles represent waiting time for an execution unit to become available.

In part *(a)* of Figure 20 the *roundrobin* policy is used for all the threads. This policy keeps a common counter value initialized to zero (the id of the first core) and every time a scheduling decision is required the current value of the counter is returned as the core identifier and the counter is then incremented by one. In part *(b)* of the figure the *dynamic* scheduling policy is used for all the threads. This policy selects the SPE core with the least load, which is the core with the least number of entries in the WQ, PWQ, PB and FQ combined, thus promoting better load-balancing. Both policies result in different schedules as shown in the figure.

Discussion

The *roundrobin* policy requires no information of the core status and aims at distributing the threads among the cores uniformly. The *dynamic* policy takes the load status in consideration

and so in the case of scheduling threads with similar execution durations, it provide optimal performance. However, when the execution duration vary greatly among the threads, an optimal schedule might not be achieved. This can be seen in the figure as the *dynamic* policy results in a longer total execution time compared to the *roundrobin* policy. To optimize the *dynamic* policy, profiling can be used to assign weights that reflect the execution duration of threads.

Specifying the scheduling policy to use with each thread and defining *custom* scheduling policies are described in detail in Chapter 5.

3.3.5 Execution Termination

Detecting termination of programs in the data-flow execution model is different from that on the control-flow model. In the latter instructions have a complete ordering that makes this task straight-forward. In the former, however, the availability of data governs the order of execution making this task more involving.

Explicit Termination Approach

The initial approach we used for detecting the termination was to designate a specific invocation of a thread as the *last* executed one in the program and once the TSU is informed of the completion of this invocation execution terminates. When splitting the graphs of big DDM programs into sub-graphs called *DDM blocks*, each block is assigned an *inlet* thread responsible for loading the *meta-data* of the block threads into the TSU, and an *outlet* thread that runs only after all the threads in the block has finished execution. The designated *last* program thread in this case would be one that is enabled when all the *outlet* threads of the program DDM blocks finish execution.

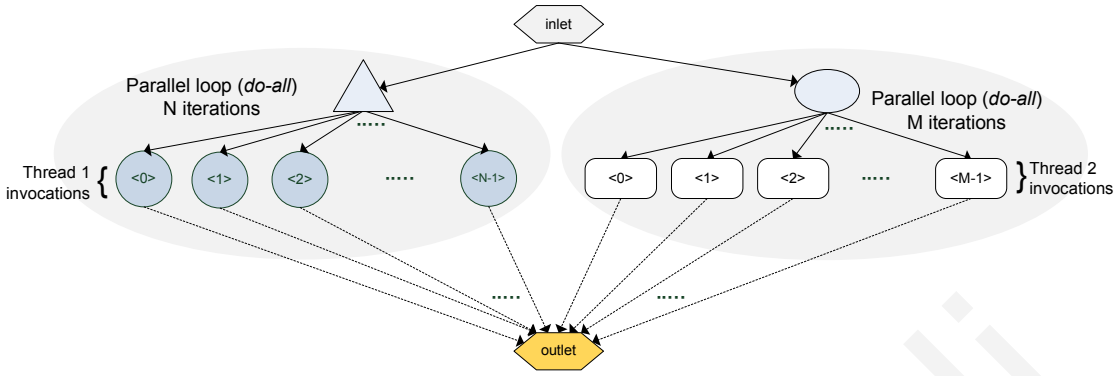


Figure 21: DDM program of two parallel loops with inlet & outlet threads

Implicit Termination Approach

In a later stage of this work, we opted for a more general and implicit approach, in which we terminate once *all the queues of the TSU are empty and there exists no pending operations in-flight*. This approach requires no change to programs using the first one as termination would still be detected after the execution of the last *outlet* thread as before.

Comparison

The advantages of the second approach are: First, it is implicit; it requires no involvement from the programmer or compilation tool to specify the thread invocation designated as the *last*. More importantly, it avoids introducing extra dependencies in the cases where such a *last* thread doesn't exist in the program. Reducing dependencies can improve the performance of the program as these extra dependencies result in extra TSU work to decrement the RCs. Figure 21 shows an example of a DDM program implementing two parallel loops with N and M iterations, respectively. To detect termination according to the first approach an *outlet* thread is introduced, however, *N+M* extra dependencies are added to the program so that the *outlet* thread executes only after all the program thread invocations finish execution. If the second approach is used, the *outlet* thread would not be required (and so the extra dependencies are not needed) and termination will still be

detected when all the threads invocations in the program finish. Moreover, in distributed DDM execution (discussed in Chapter 4), the extra dependencies required by the first approach might generate extra network messages as the invocations might be executing across all the nodes in the system. One shortcoming of this approach is that when it is utilized, it is difficult to distinguish between successful completion and abnormal termination.

On the other hand, the first approach is more appropriate in the cases where multiple DDM applications are to be scheduled by the same TSU and so detecting the termination of each application separately is desirable. Furthermore, in certain cases (as will be demonstrated in Chapter 5) extra dependencies are introduced to control the amount of parallelism exposed in the program, in these cases using the first approach comes at almost no cost.

The DDM-VM supports both approaches: if an arbitrary thread invocation is specified as the *last* one, termination occurs when that invocation finish, otherwise termination is decided according to the second approach. Selecting the approach to use is based on the properties of the application.

3.4 Software CacheFlow (S-CacheFlow)

In this section we present the design and implementation details of the Software CacheFlow (S-CacheFlow), a prefetching software cache developed as part of the TSU for the management of the memory hierarchy in the DDM-VM$_c$. We describe the structures and operations of the S-CacheFlow and illustrate the various techniques and optimizations we implemented for improving the performance.

Multi-core architectures with software-managed on-chip memories [55, 65, 63, 30, 127, 54] introduce private address spaces and rely on software to manage data transfers, which can be both complex and error-prone [105]. This tasks is more challenging on the Cell because the LS is a constrained memory resource demanding efficient utilization. Moreover, having the LS as software-controlled renders many techniques applied to preserve coherency in hardware-caches prohibitively expensive. To handle these challenges, DDM-VM$_c$ utilizes the *CacheFlow* [72] policy to implement Software CacheFlow (S-CacheFlow): a fully automated prefetching software cache with variable cache block sizes that is extended with locality optimizations. Moreover, S-CacheFlow maintains consistency using data-flow synchronization in a technique similar to DAG Consistency [20]; it prefetches input data from main memory to the LS before the thread starts execution and writes-back produced data to main memory after the thread finishes execution. S-CacheFlow support *explicit locality*, which avoids expensive cross-SPE coherence operations.

CacheFlow is a data-driven cache management policy utilized with DDM to improve the performance by ensuring that the data a thread requires is in the cache before the thread is fired for execution. The original implementation of CacheFlow [72] targeted machines with hardware

caches to implicitly improve the performance of DDM execution by reducing cache misses. However, in this work CacheFlow is applied in a new context, that is, to manage the memory hierarchy in multi-core architectures with software-managed memories like the Cell.

Next, we describe the structures and operations of the S-CacheFlow module in the TSU.

3.4.1 S-CacheFlow Structures

To implement S-CacheFlow on the Cell a portion of the LS memory of each SPE, typically (96-128)KB, is pre-allocated and divided into cache blocks. We refer to this portion as the *DDM Cache*. The size of the blocks can vary to match each application characteristics but must be in multiples of 128B, which is the cache-line size and the minimum granularity for DMA transfers on the Cell processor. Note that transfers of smaller sizes are allowed, nevertheless, a bus bandwidth of a full 128B is consumed. Therefore, DMA throughput is maximized if transfers are at least 128B [5].

The per-SPE S-CacheFlow structures allocated in main memory consist of the Cache Directory (CD) and the The Remote Cache Lookup Directory (RCLD). The structures are depicted in Figure 17.

3.4.1.1 The Cache Directory (CD)

The CD holds the state of the cache blocks. It maps an address range in the main memory address space into one in the LS private address space. Each input/output data of a thread is allocated at least one cache block and data instances larger than one cache block are allocated in consecutive blocks. This setup simplifies all the CD operations and improves the performance of the cache.

Each entry corresponds to one cache block or more (depending on whether the *size* field is equal to 1 or more). Each entry is composed of the following fields:

- **Valid**: indicates if the block(s) contain valid data.
- **EAddr**: the main memory address (*Effective Address* in the Cell terminology) of the block(s) data .
- **Size**: the size of the data associated with this entry in terms of cache blocks.
- **Dirty**: indicates if the block(s) data has been modified when kept in the LS for re-use.
- **UserNum**: the number of threads (waiting to execute or currently executing) that are referring to the cache block(s).
- **reference-count**: value needed for exploiting locality, it specifies the number of remaining re-uses of the block(s) before they are flushed to main memory or invalidated.

Note that the index of each entry indicates the cache block number in the *DDM cache* and consequently its LS address.

3.4.1.2 The Remote Cache Lookup Directory (RCLD)

The RCLD holds the lookup information needed by the runtime on the SPE. The most important information is the LS address where the data was allocated. Each RCLD entry is composed of the following fields:

- **TID**: ThreadId of the thread whose data is associated with this entry.
- **context**: the *context* specifying the exact invocation of the thread.
- **DFPIndex**: indicates for which DFP this data belongs (the first, second, etc.)

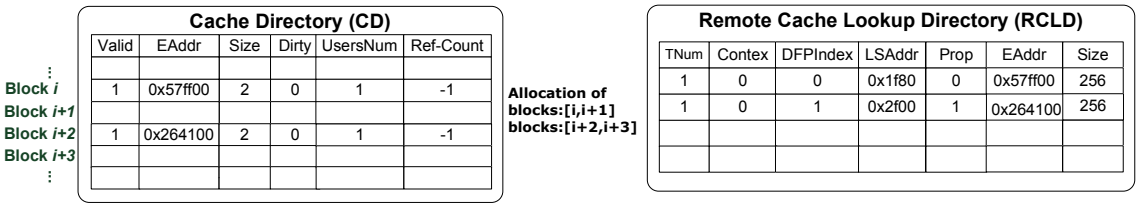


Figure 22: S-CacheFlow Allocation Example: the contents of the Remote Cache Lookup Directory (RCLD) & the Cache Directory (CD)

- **LSAddr**: the address of the data in the LS
- **Prop**: properties of the data. This indicates if the data access is for reading, writing or both and whether the data associated with this entry must be written-back to main memory after the thread finishes execution.
- **EAddr**: the main memory address of the data. This is required for writing-back produced data to main memory after the thread finishes execution.
- **Size**: the size of the data in bytes. This is required for writing-back produced data to main memory after the thread finishes execution.

Allocation Example

Figure 22 illustrates the state of the CD and RCLD after the data allocation for a DDM thread that requires reading data at main memory address 0x0057ff00 of size 256 bytes and writing data at main memory address of 0x00264100 of size 256 bytes. In this example the cache block size is assumed to be 128B and the data is allocated in blocks:[i,i+3] of the cache.

3.4.2 S-CacheFlow Operations

In this section we describe the S-CacheFlow operations. We logically divide the operations into those preceding a thread execution and those after.

3.4.2.1 Pre-Thread Execution

TSU Operations

At runtime the S-CacheFlow module in the TSU dequeues the entries of threads from the WQ and retrieves the information of the thread input/output data by calling a helper-function. The retrieved information includes a list of DFP tuples describing the input/output data. The tuple consists of the following elements:

- Main memory address
- Size
- Properties

The properties specify whether the data access is for reading, writing or both, in addition to other flags and the reference-count used for exploiting data locality. S-CacheFlow uses the retrieved information to allocate the thread data in the *DDM Cache* at the SPE where the thread is scheduled to run by consulting the CD. If the allocation is successful, the LS addresses of the allocated cache blocks are returned. The returned addresses along with the tuples list constitute all the information needed for both data fetching and -later- cache lookup.

In particular, S-CacheFlow uses this information to issues DMA calls for transferring the data from main memory to the LS by placing requests in the Proxy Command Queue of the MFC of the target SPE. The RCLD directory entries in main memory are populated and transferred via DMA calls to the LS. The issued DMA calls (for both data and RCLD entries) pertaining to the same thread are assigned a unique 5-bit *tag* that is stored in association with the thread information in a special buffer called the PendingBuffer (PB). The status of the issued DMA calls in the PB are checked periodically for completion via special synchronization instructions that make use of the

tag. Any thread whose DMA calls are completed is moved from the PB into the FQ indicating that it is ready for execution. Figure 23 depicts this part of the S-CacheFlow operations.

SPE Runtime Operations

The runtime on the SPE fetches the information of the next thread to execute from the FQ and then uses the ThreadId and *context* of the thread as keys to find the associated RCLD entries so as to assign the pointers used to access the data. This is required because data belonging to different threads can be present in the LS due to prefetching. Note that because the number of the RCLD entries is limited, this cache lookup operation is very quick and efficient.

3.4.2.2 Post-Thread Execution

SPE Runtime Operations

After the thread finishes execution its output (modified) data is written-back to the main memory via DMA calls to maintain data consistency. However, when exploiting data locality modified data is not written-back immediately, but rather kept in the LS to be re-used by consumer threads scheduled to run on the same SPE core. A special flag in the *Prop* field of the RCLD entries of output data informs the runtime whether to write-back this data to main memory or not. A detailed discussion of exploiting locality in S-CacheFlow is presented in 3.4.4.

TSU Operations

When the TSU is notified that a thread finished execution, the S-CacheFlow performs a number of house-keeping tasks. It decrements the *UserNum* counter in the CD entries associated with the cache blocks allocated by the thread. If the *UserNum* counter of a certain entry reaches zero, the associated blocks are considered free and the CD entry is invalidated (the valid and dirty bits are

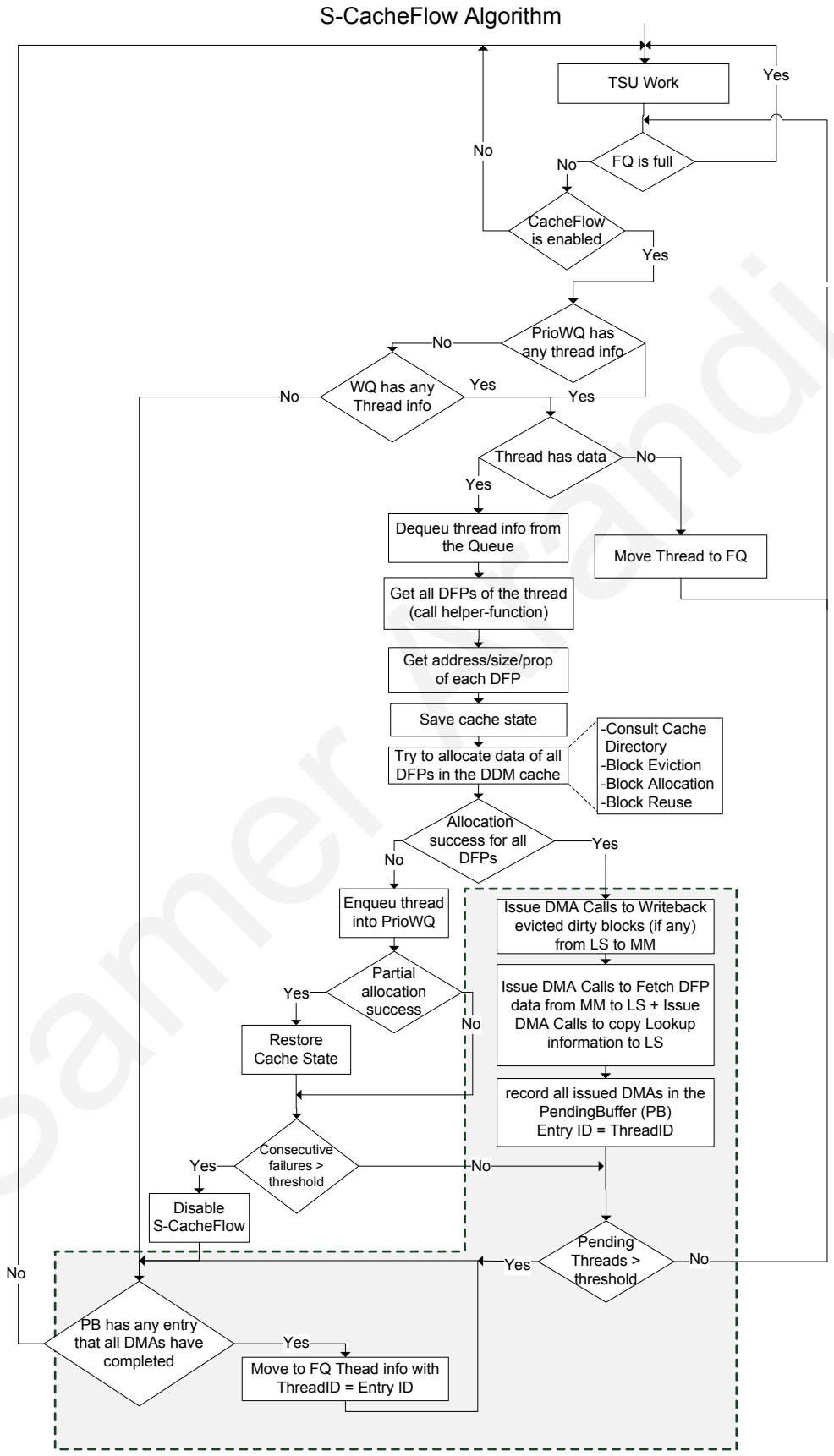


Figure 23: S-CacheFlow Algorithm - Pre-Thread Operations (shaded parts are executed on the SPE in the Distributed S-CacheFlow implementation)

cleared). This default behaviour is overridden in the case of blocks that are kept in the cache for re-use to take advantage of locality. Finally, the entries of the RCLD associated with the thread are also cleared.

In the following section we describe the S-CacheFlow allocation and eviction procedures in more detail.

3.4.3 Allocation and Eviction

3.4.3.1 Allocation Procedure

Each input/output data of a thread is allocated at least one cache block. S-CacheFlow checks first if the address of the data in main memory is properly aligned according to the following alignment requirements and guidelines of the Cell:

- DMA transfers of 16B or greater must be aligned on a 16B boundary (both the local store and main memory addresses)
- DMA transfers of 1, 2, 4, or 8 bytes are supported but must be *naturally aligned* (the starting local store and main memory addresses must be divisible by the size of the transfer and have the same local store and memory address offsets with a 16-byte block).
- DMA throughput is maximized if transfers are at least 128B, and transfers greater than or equal to 128B should be aligned to 128B.

Typically, Cell programmers are encouraged to aligned their data on 128B boundaries and since S-CacheFlow allocates data in blocks of 128B or multiples of this size, all the alignment requirements are satisfied and the optimal performance is achieved. However, if the user data is not aligned properly, the data address pointers are fixed to start at the nearest 128B aligned address.

The actual start of the data is recorded along with the actual size and used later when populating the RCLD entries.

After the alignment step, S-CacheFlow checks if the data *re-use* optimization (described in the next section) is enabled for this input data and if so, it tries to find if the data already exists in the LS by looking up the address in the CD. If the address is found (cache hit), it is returned and no allocation takes place. If the address is not found (cache miss) or re-use is not enabled, S-CacheFlow tries to find free cache blocks to allocate.

Locating Free Blocks

To optimize the process of finding free cache blocks an auxiliary bitmap directory is maintained as an array of bytes. Each bit corresponds to the state (free or allocated) of one cache block in the LS. Due to its smaller size (compared to the CD) the bitmap directory allows finding free blocks more efficiently. For example, if the value of one byte in the array is zero, it indicates the availability of the corresponding eight consecutive free cache blocks. Naturally, all operations on the CD are reflected on the bitmap directory to keep it updated.

3.4.3.2 Eviction Procedure

When the allocation procedure fails to find free blocks, a simple cache eviction policy is invoked. The policy prohibits evicting any cache block that has the *UserNum* in the associated CD entry greater than zero. This guards against evicting cache blocks of threads that are currently executing or in the FQ. Note that in the original implementation of CacheFlow eviction of such blocks would have caused a cache-miss affecting the system performance, but in the Cell implementation the result would be an invalid data in the cache.

In the case of evicting dirty block(s) (the *dirty* field in the CD is set), the dirty block(s) address is recorded. Later on, at the data fetching stage, additional DMA calls are inserted to write-back those cache blocks from the LS to main memory before the DMA calls fetching the data have effect. Enforcing this ordering is done using *fencing* synchronization commands. If no blocks are found for eviction the allocation fails. Figure 24 illustrates an overview of the allocation and eviction procedure.

3.4.3.3 Allocation Failure

If S-CacheFlow fails to allocate data for any of the inputs/outputs of a thread, it is not processed further and another thread in the WQ is processed. However, to avoid thread starvation this thread is inserted into the Priority Waiting Queue (PWQ) which has a higher-priority and so will be checked first by the TSU on the next TSU cycle.

In the case of partially successful allocations (where only a subset of the thread data has been allocated due to LS space shortage) the effect of the partial allocations on the state of the S-CacheFlow must be rolled back. To this end, S-CacheFlow keeps track of all the changes made by the allocation and eviction procedure in a special *log buffer* that is used to revoke all changes in case of failure. The buffer is flushed upon successful allocation of all the data of a thread. When the consecutive allocation failures reaches a certain threshold (typically occurs when the LS is full), the S-CacheFlow module is disabled to reduce its overheads. It is only enabled when a thread finishes execution and its associated LS memory is freed.

3.4.4 Exploiting Data Locality

S-CacheFlow exploits locality whenever multiple threads invocations scheduled to execute on the same SPE access the same data, by keeping the blocks of this data in the LS instead of writing

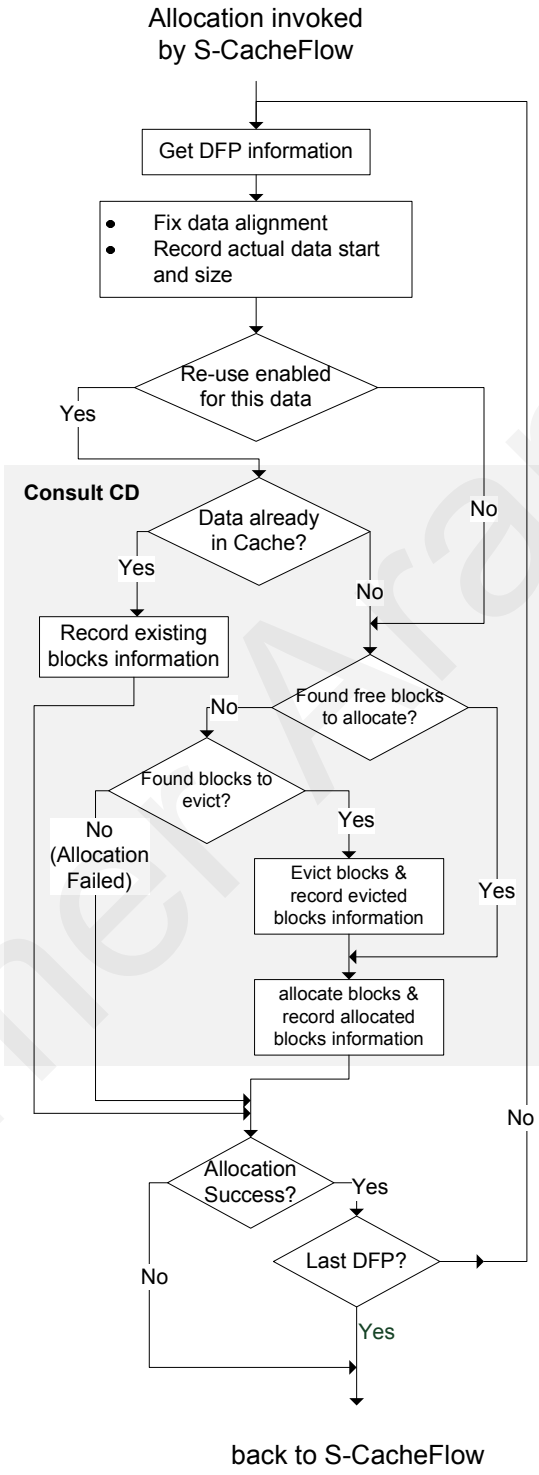


Figure 24: Allocation and eviction in the S-CacheFlow algorithm)

it back to main memory. In addition to the benefit of saving the memory bandwidth, this can result in conserving the LS space, as the data of more threads can fit simultaneously in the LS if such threads share the same input data. This increases the amount of parallelism and improves the system performance.

The cases that benefit from data locality are: (i) data produced by a *producer* thread and kept in the LS to be re-used by *consumer* threads accessing this data for reading and scheduled to run on the same SPE or (ii) data located at main memory (either produced during the initialization stage or produced by a thread and written-back to main memory) and fetched into the LS for read access by a thread and then kept there to be re-used by other threads reading the same data.

3.4.4.1 The Data Re-use Mechanism

Producer Thread Side

In either of the two cases utilizing data locality, a special flag *(DATA_KEEP)* is set in the properties of the DFP associated with the producer thread data that is expected to be re-used. In the first case, the flag causes the S-CacheFlow to instruct the runtime on the SPE to not write the data back to main memory. This is achieved by setting the same flag in the *Prop* field of the RCLD entry associated with the data. Moreover, the *dirty* field of the CD entry associated with the data is set to make sure the data will be written-back to memory at the end of the program or when the blocks are selected for eviction. Finally, when the thread finishes execution the CD entry is not invalidated. Similarly, in the second case the CD entry is not invalidated when the thread finishes execution, however, the *dirty* field is not set.

Consumer Thread Side

The *consumer* threads expected to re-use the data have another special flag *(DATA_REUSE)* set in the properties of the DFP of the consumed data. The flag causes the S-CacheFlow to perform a lookup on the CD at the time of allocation as described previously in Section 3.4.3. If the lookup operation results in a hit, the address of the re-used data in the LS is returned and no allocation or fetching of data occurs. Additionally, the *UserNum* field in the CD entry associated with the data is incremented and only decremented when the thread finishes execution.

This *explicit locality* scheme minimizes overheads typically associated with software caches as only data with the *DATA_REUSE* flag results in expensive lookup operations on the CD. Moreover, since this lookup will be a hit in most cases, the overhead is offset by the benefits of the re-use.

3.4.4.2 Preserving Consistency

When employing locality care must be taken in some of the cases where it is necessary to specify ”when” to write-back data kept for re-use to main memory to guarantee cross-SPE coherence. For example in the case we keep produced data that (after an arbitrary number of re-uses) is needed by consumer threads running on different SPEs. To handle this, the following two techniques can be used:

- explicitly inserting ”writes” in the graph of the program to make sure data is written-back to main memory before it is required by any consumer running on a different SPE.
- assigning a *reference-count* value for every such data to ensure that the data is written-back to main memory after a specific number of ”re-uses” on the current SPE.

Setting the Reference-count

Since the main goal is to write-back the produced data before a consumer on another SPE reads it, we set the value of the *reference-count* to *the number of expected reads occurring on the current SPE before a read will occur on any other SPE.*

In the cases where we allow in-place-updates i.e. writing to the same address more than once (to optimize accumulation operations) we have an additional concern: ensuring that when a write occurs on an SPE, no valid copy of the data exists anywhere. In this case the value of the *reference-count* is set to *the number of expected reads occurring on the current SPE before a read or write occurs on any other SPE.*

Figure 25 shows an example of a simple DDM-VM program utilizing locality. The program is composed of 4 threads. The first three threads are mapped to SPE0 and the last to SPE1. Thread *T1* produces a value *A* that is consumed by all the iterations of thread *T2*, in addition to threads *T3* and *T4*. The figure also illustrates the data transfers between the LS and the main memory that occur before or after every thread finishes execution, as a dashed arrow to the right of the thread. In part (a) of the figure, data re-use is not exploited and so once *A* is produced by *T1*, it is written back to the main memory and for each iteration of *T2*, *A* is fetched from main memory every time and the same applies to *T3* and *T4*. In part (b) of the figure data re-use on the value *A* is enabled by adding the *KEEP* flag for the corresponding DFP of *T1* and the *REUSE* flag for the corresponding DFPs of T2 and T3. Note that the *reference-count* value is set to *N+1*, which is the number of expected reads/re-uses on SPE0 before the read on SPE1 (by T4) occurs. This guarantees that *A* is written back to main memory once *T3* finish execution and so *T4* will read the correct value of *A* from main memory. It is clear that utilizing data re-use in this example eliminates *N+1* transfers from main memory to the LS.

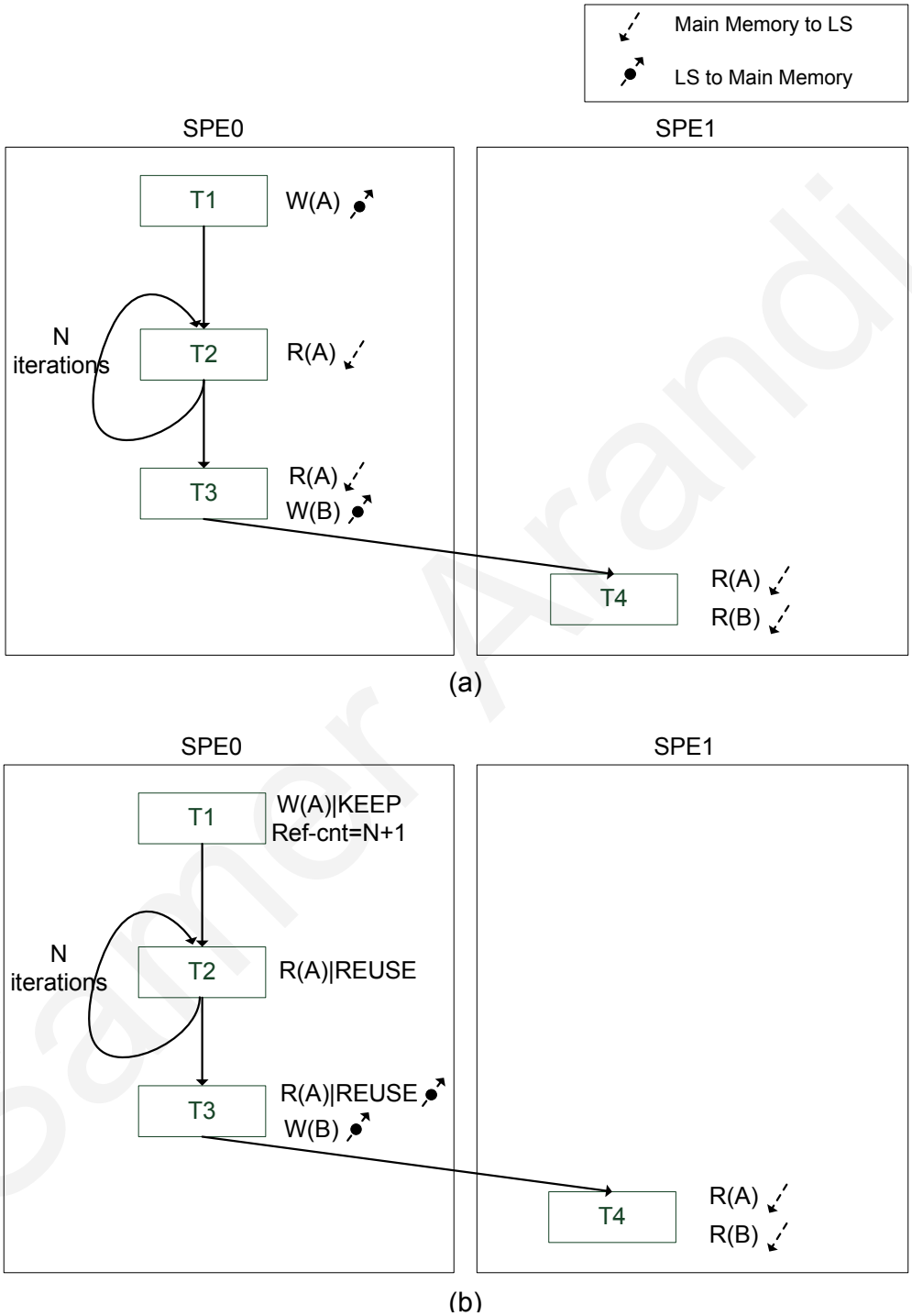


Figure 25: An Example of a DDM-VM program Utilizing Locality. (a) no locality (b) with locality

Discussion

The drawback of the *reference-count* technique is that if a cache block is evicted before its *reference-count* has expired due to the eviction policy, its *reference-count* must be saved in a special buffer in the TSU and restored when the block is brought back again at the next re-use. Moreover, this technique is conservative; if we cannot determine the value of the *reference-count* at compile time using the measures we mentioned, we don't apply it. If no *reference-count* is assigned when using re-use, S-CacheFlow reverts to the first technique i.e. it assumes that the programmer inserted the appropriate "writes" wherever needed.

At the end of program execution the S-CacheFlow module checks the CD of all the SPEs and flushes all dirty block to main memory.

We have found that in many cases exploiting locality requires utilizing neither technique as the threads in these cases re-use data that was initialized at startup and not modified again or data produced and consumed on the same SPE and so it was enough to be flushed at the end of execution by the runtime. In such cases it was merely enough to add the *DATA_KEEP* and *DATA_REUSE* flags appropriately and the rest was managed by the VM.

3.4.4.3 Preserving the Allocation Order

Exploiting locality requires that the order in which threads have their data allocated and the order in which the threads are executed be the same. This is necessary in the cases where a thread *B* re-uses data that will be fetched to the LS by a thread *A* that finished its data allocation before *B* but hasn't executed yet. If the execution order is changed and *B* executes before *A*, it would access data that hasn't been fetched to the LS yet. The ordering is achieved by assigning a *sequence number* to threads according to the order they are inserted in the PB and checking this sequence

before executing the thread on the SPE. This ordering only applies to execution and doesn't affect prefetching, i.e. DMA calls fetching data can be issued out-of-order.

3.4.5 Distributed CacheFlow

The evaluation of the initial implementation of the DDM-VM$_c$ & S-CacheFlow scaled well for up to 4 SPE cores, but for a higher count of cores the PPE became a bottleneck. Our analysis revealed that a major source of overhead was the tasks of issuing a large number of DMAs and periodically checking their completion, which overloaded the PPE core. The reason is that the PPE had to communicate with the MFC to perform these tasks. This type of communication is implemented through the Memory-Mapped I/O (MMIO) interface and hence is expensive. To solve this problem we have modified the S-CacheFlow implementation and moved the DMA management to the portion of the runtime that runs on the SPEs. This proved more advantageous as the SPEs are more efficient at enqueuing DMA requests (smaller issuing latency and less overhead on the internal bus). Furthermore, the DMA command queue holding SPE-initiated DMAs is twice as deep compared to the proxy MFC command queue holding PPE-initiated DMAs [5]. Finally, distributing this task on the cores improves the scaling and reduces the pressure on the PPE core. We refer to this new implementation as the Distributed S-CacheFlow (not to be confused with distributed DDM execution on multiple nodes). Evaluation of both implementations is presented in the Evaluation Chapter and the results clearly show the advantages of the distributed implementation. Next, we describe the Distributed S-CacheFlow implementation by highlighting the changes that was made to the TSU and the TSU-Processor interface in due.

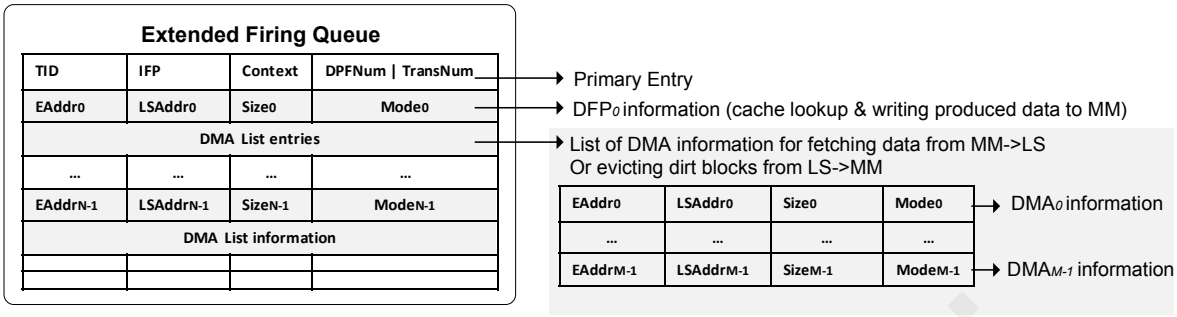


Figure 26: The extended FQ

CacheFlow Structures

The PB and FQ are moved to the SPE runtime structures allocated in the LS. The RCLD is extended to record the information of the DMA calls in addition to its original function of holding the LS data pointers. In a final refinement we merged the FQ with the RCLD to avoid storing redundant information given the limited size of the LS. Each entry in the resulting structure that we call the Extended FQ (ExFQ) holds a primary sub-entry recording the thread information (ThreadId, IFP, *context*, number of DFPs and number of transfers), followed by secondary sub-entries recording the cache lookup information for each input/output in addition to all the DMA information for that input/output (source address, destination address, size, flags). Remember that one input/ouput data could require multiple DMA calls. Figure 26 depicts the new ExFQ.

S-CacheFlow Operations

Under the new implementation, threads in the WQ are processed by the S-CacheFlow module that performs all the tasks of allocation as before, however, instead of issuing the DMA calls it only inserts an ExFQ entry with the primary and secondary information. The TSU then issues a DMA call to transfer the ExFQ entries to the SPE and another one to set a flag in the LS. A fencing operation ensures that the flag is only set after the first DMA call completes. The runtime on the SPE periodically checks the flag and when it is set, it accesses the corresponding ExFQ

entry information, it issues the DMA calls using a unique *tag* for the DMAs pertaining to that entry and adds the *tag* to the PB. The tags in PB are checked periodically and when the DMAs complete the corresponding ExFQ entry is consulted again to get the IFP, *context* and the LS pointers information and the execution of the thread starts. When a thread finishes execution activities proceed as before without a change, except that the runtime consults the ExFQ instead of the RCLD when writing produced data to main memory. Figure 27 depicts the activities of the SPE runtime.

3.4.6 Adaptive Multi-buffering/Prefetching

Multi-buffering is a technique in which the programmer partitions the application data into multiple sets and then re-organize his code to work on one set while asynchronously fetching the next one(s) from main memory. This interleaving between computation and data transfer is one of the main techniques utilized in the Cell to overcome the Memory Wall. This is made possible by the ability of the MFC units to issue a DMA call and check their completion asynchronously. However, a considerable effort from the programmer is required to structure his code to take advantage of multi-buffering.

S-CacheFlow takes advantage of the MFC facilities to issue multiple DMAs for the data belonging to multiple threads without waiting for the transfers to complete. This allows the prefetching of the data whenever possible, and hides the latency of the data transfers with the execution of other threads. Therefore, it effectively achieves an automatic and transparent multi-buffering that adapts to the number of ready threads and the LS space limitation.

Moreover, when a thread finishes execution the DMAs copying the DDM commands to the CQ and the ones writing-back produced data to main memory are issued asynchronously and the execution of the next ready thread commences without waiting for the DMAs to finish. Thus,

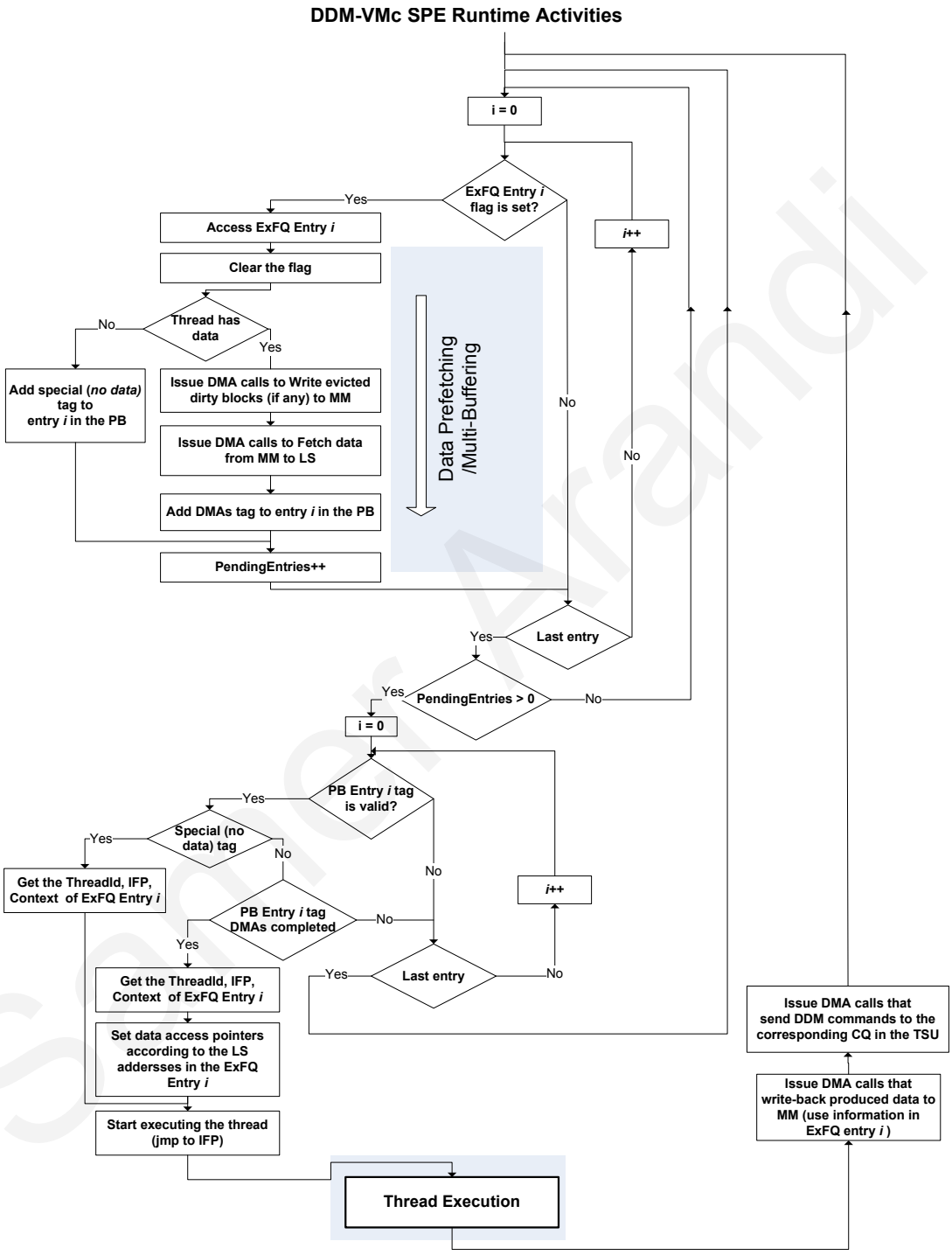


Figure 27: DDM-VM$_c$ SPE runtime activities

overlapping the LS to main memory DMA transfers with execution to further tolerate latencies. The only requirement is that the data transfers DMAs complete before the ones copying the DDM commands that notify the TSU that the thread finished execution. This is achieved using *fencing* synchronization commands that enforce this ordering.

Figure 28 demonstrates the benefits of prefetching/multi-buffering and the overlapping of DMA transfers with the execution of threads by comparing the execution time for four threads with and without prefetching. The figure shows that when utilizing prefetching/multi-buffering the total execution time decreases as the latency of DMA transfers (for both data and commands) is overlapped with the execution of the threads. The figure assumes that after executing a thread, there exist at least one thread (whose data is ready for prefetching) in the ExFQ.

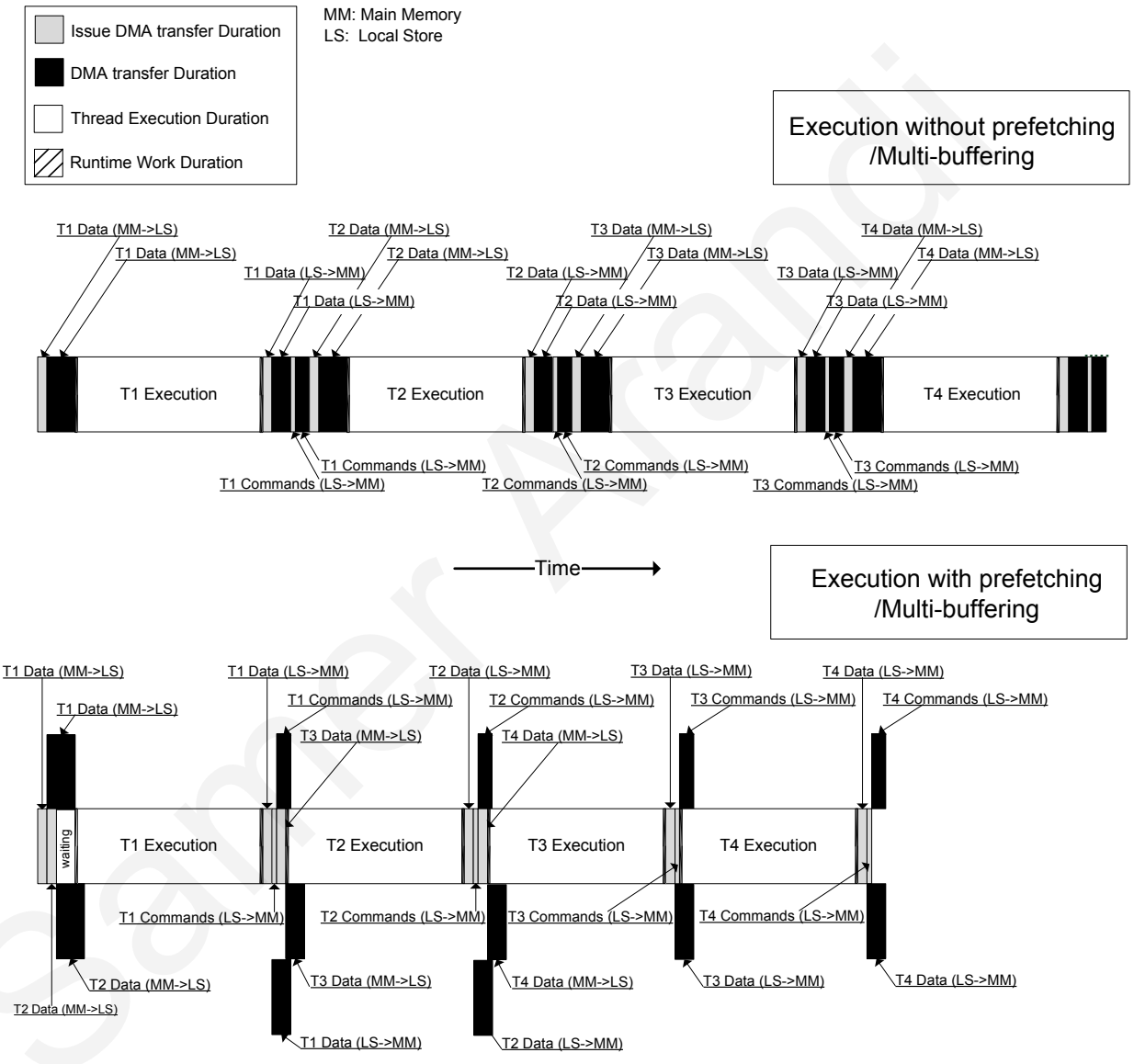


Figure 28: Comparison of execution time with and without prefetching

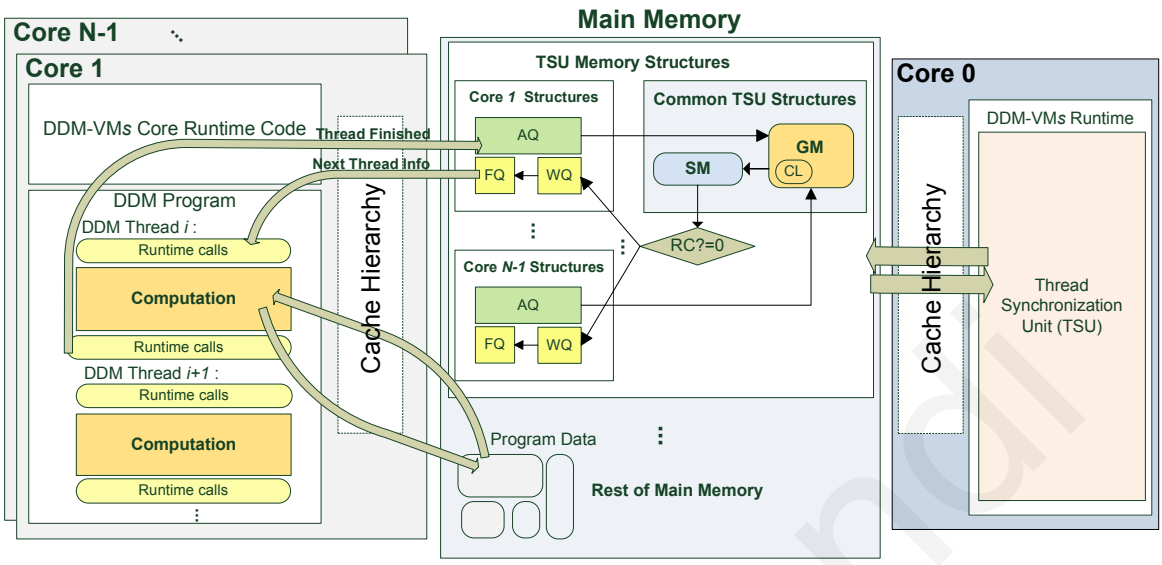


Figure 29: The Architecture of the DDM-VM$_s$

3.5 The Data-Driven Multithreading Virtual Machine for Symmetric Multi-cores (DDM-VM$_s$)

The DDM-VM$_s$ is the DDM-VM implementation targeting homogeneous multi-core systems. The DDM-VM$_s$ is composed of two main parts: The first is the TSU, which runs as a software module on one of the cores. The second consists of the runtime threads spawned on the rest of the cores to manage the execution of the DDM threads and the communication with the TSU. The communication is performed by changing the state of the TSU structures allocated in main memory. Figure 29 illustrates the architecture of the DDM-VM$_s$.

The task of implementing the DDM-VM$_s$ was accomplished by using the main part of the DDM-VM$_c$ implementation and adapting it for homogeneous multi-core architectures. This effort was accomplished in collaboration with George Michael as the goal of his undergraduate thesis [83]. Adapting the DDM-VM$_c$ to implement the DDM-VM$_s$ was possible as the two implementations share the basic functionality of the TSU. The main differences between the two implementations stem from the differences between the two underlying targeted architectures. The TSU in

the DDM-VM$_c$ implementation runs on the PPE core which has a separate address space from the SPE cores executing the threads. This is in contrast with the the DDM-VM$_s$ where all the cores share the same address space. Moreover, the DDM-VM$_c$ implements a prefeching software cache for managing the memory hierarchy of the Cell, while the memory hierarchy is managed by hardware in the DDM-VM$_s$. These differences prompted the modification of parts of the internal implementation of the TSU in addition to changing the interface between the execution cores & the TSU.

In the next section we describe briefly the structures and operations of the TSU and the TSU-Processor interface and highlight the differences in comparison with the DDM-VM$_c$ implementation.

3.5.1 The Thread Scheduling Unit (TSU)

The TSU Memory Structures

The TSU structures are allocated in main memory. Part of the structures is common for all the cores and the rest are per core. The common structures include: the **Graph Memory (GM)**, the **Consumer List (CL)** and the **Synchronization Memory (SM)**.

The per-core structures include: the **Acknowledgement Queue (AQ)**, the **Waiting Queue (WQ)** and the **Firing Queue (FQ)**. Note that the **Command Queue (CQ)**, the **Priority Waiting Queue (PWQ)** and the **Pending Buffer (PB)** along with the structures required for the operation of the S-CacheFlow are not needed in the case of the DDM-VM$_s$. Note that the AQ is allocated per-core and the FQ has been extended to record the information of the thread input/output data. This information is needed for supporting distributed DDM execution, in which part of the thread data could be allocated dynamically by the TSU (as will be described in the following chapter).

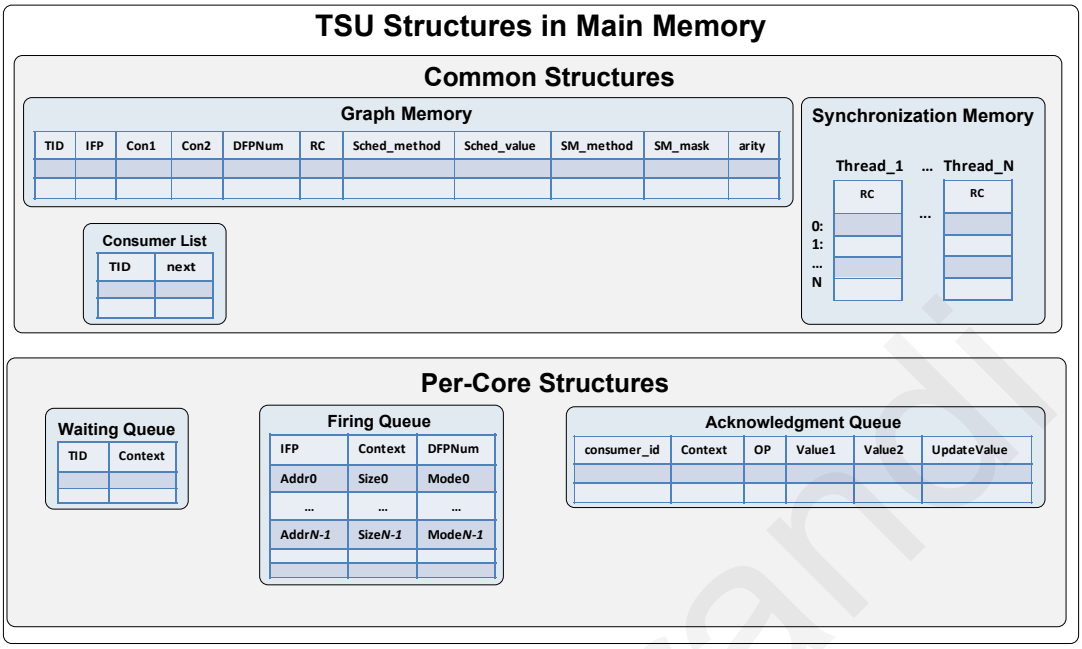


Figure 30: The TSU Structures in the DDM-VM$_s$

Figure 30 depicts the structures of the TSU. For a full description of the information held in the structures refer to 3.3.2.

DDM Thread Execution

The execution of the DDM threads takes place on the system cores as described previously in 3.3.2.1. The only difference is related to the TSU-Processor interface used by the runtime to notify the TSU that a thread finished execution and to retrieve the information of the next ready thread. This is presented in the next section.

The TSU Operations

The TSU runs on one of the cores and operates as described previously in 3.3.2.1, however, the TSU processes the AQ entries without first executing the commands in the CQs, as the CQs is not needed in the DDM-VM$_s$. Furthermore, the S-CacheFlow data allocation and eviction

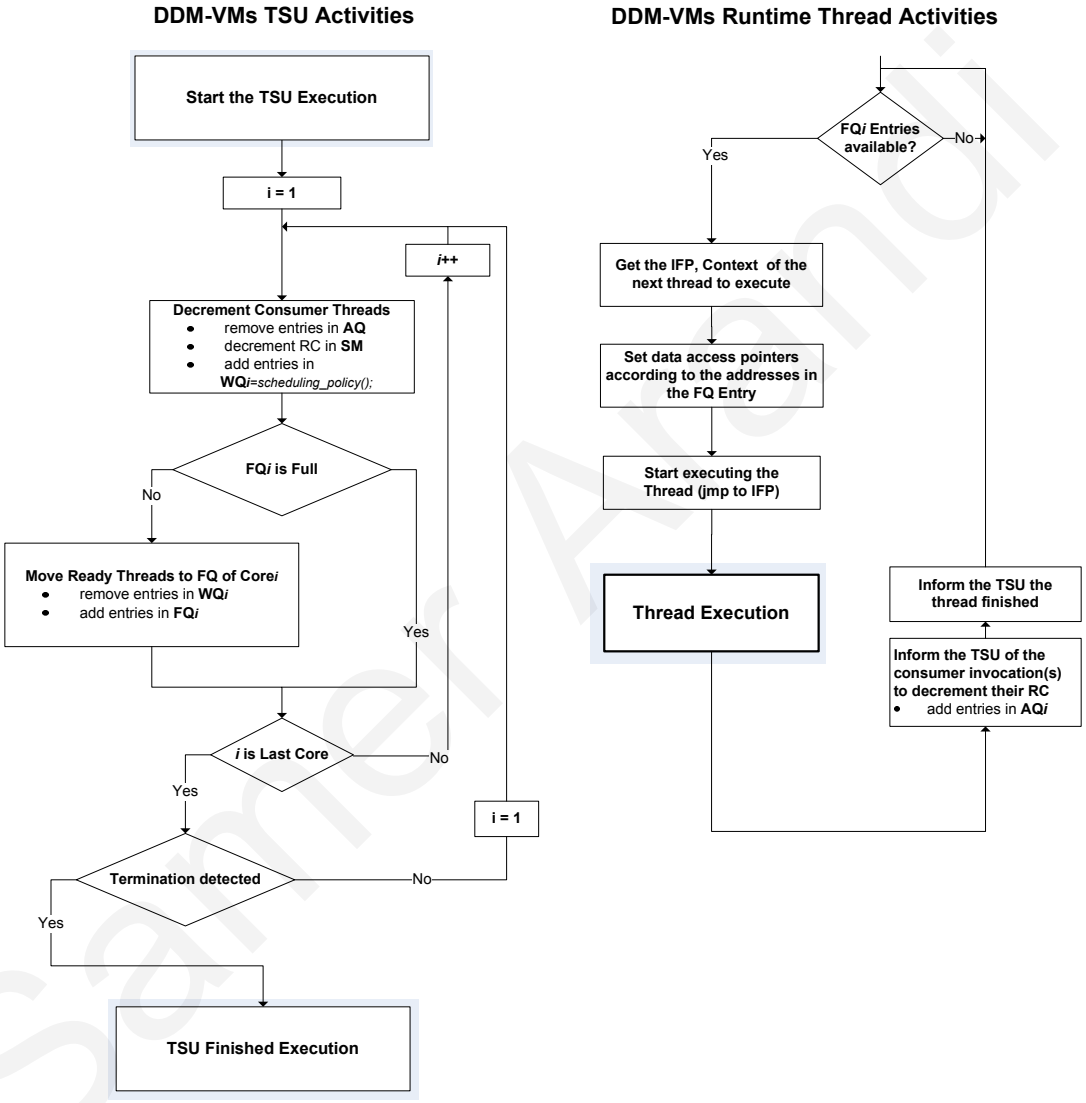


Figure 31: The DDM-VM$_s$ TSU and Runtime Activities

operations that take place before moving a thread from the WQ to the FQ is not performed since those operations are managed by the processor memory controller hardware in the DDM-VM$_s$. Figure 31 illustrates the activities of the TSU and the runtime threads in addition to the effects of each activity on the TSU structures.

3.5.2 TSU-Processor Interface

As a reminder, we list the main tasks performed by the TSU-Processor interface:

1. Informing the TSU that the currently executing thread has finished execution and specifying the consumer thread(s) invocation(s) to decrement their RC
2. Providing the execution cores with the information of the next ready thread to execute

The two tasks are implemented by accessing the related TSU structures in main memory directly: The AQ in the first task and the FQ in the second task. Because this access is concurrent a mechanism that keeps the state of the TSU structures consistent is required. We describe this mechanism next.

3.5.3 Handling concurrent access of the TSU structures

The two TSU structures that required protection due to concurrent access by both the TSU and runtime threads are the AQ and the FQ. Initially we resorted to adding a lock on each access to the two queues. However, due to the observed overheads of the locking mechanism we utilized a number of optimizations the resulted in removing the locking completely. The optimizations are based on the following observation: In almost all the cases (except for one in the AQ) the only conflict that arises between the TSU and each of the runtime threads when accessing the *head* or *tail* queue pointers in both structures, is a reader/writer conflict. This type of conflict allows

utilizing an optimization that avoids locking altogether at the expense of losing a small part of the queue space.

Optimizing the FQ Access

The *head* of the FQ pertaining to a certain core is only accessed by the TSU for reading, when checking the size of the FQ before inserting a new entry. On the other hand, the runtime thread on that core accesses the *head* for reading/writing when removing an entry from the FQ. The key insight is that even if we don't lock when accessing the *head*, the worst case would be the TSU reading an *old* value of the *head* and consequently assuming a size of the FQ that is less than the actual one. This assumption lasts for the duration of one TSU cycle (one complete check of all the TSU structures) before the updated *real* value is eventually seen by the TSU. The effect of this momentarily assumption on performance is negligible compared to the benefit gained by removing the locking operation. The same observation applies to the *tail* of the FQ except that it is the runtime thread this time that potentially sees an older value. A similar optimization technique was employed by the TFlux platform [113].

Optimizing the AQ Access

Originally we utilized one common AQ for all the cores (similar to the design of the DDM-VM$_c$). This results in a multiple-writers/single-reader conflict on the common *head* pointer (the multiple writer being the runtime threads from all the cores). This type of conflict precludes us from applying the previously mentioned optimization and so a lock is unavoidable. To resolve this, we split the AQ internally into multiple AQs, one per core. Hence, the runtime thread on each core writes to its associated AQ and the TSU processes multiple AQs instead of only one. This enabled us to apply the same optimization as the FQ and remove the locking operations.

Chapter 4

Distributed Data-Driven Execution

In this chapter we describe the extension of the DDM-VM to support DDM execution across a number of multi-core nodes (a cluster) connected over an off-chip network. Each node is an independent multi-core machine running an operating system and capable of executing multiple DDM threads concurrently. A Shared Global Address Space is supported across all the nodes in the system. Figure 32 illustrates an overview of the distributed DDM-VM architecture. This chapter highlights the extensions and modifications of the DDM-VM design to support distributed DDM execution.

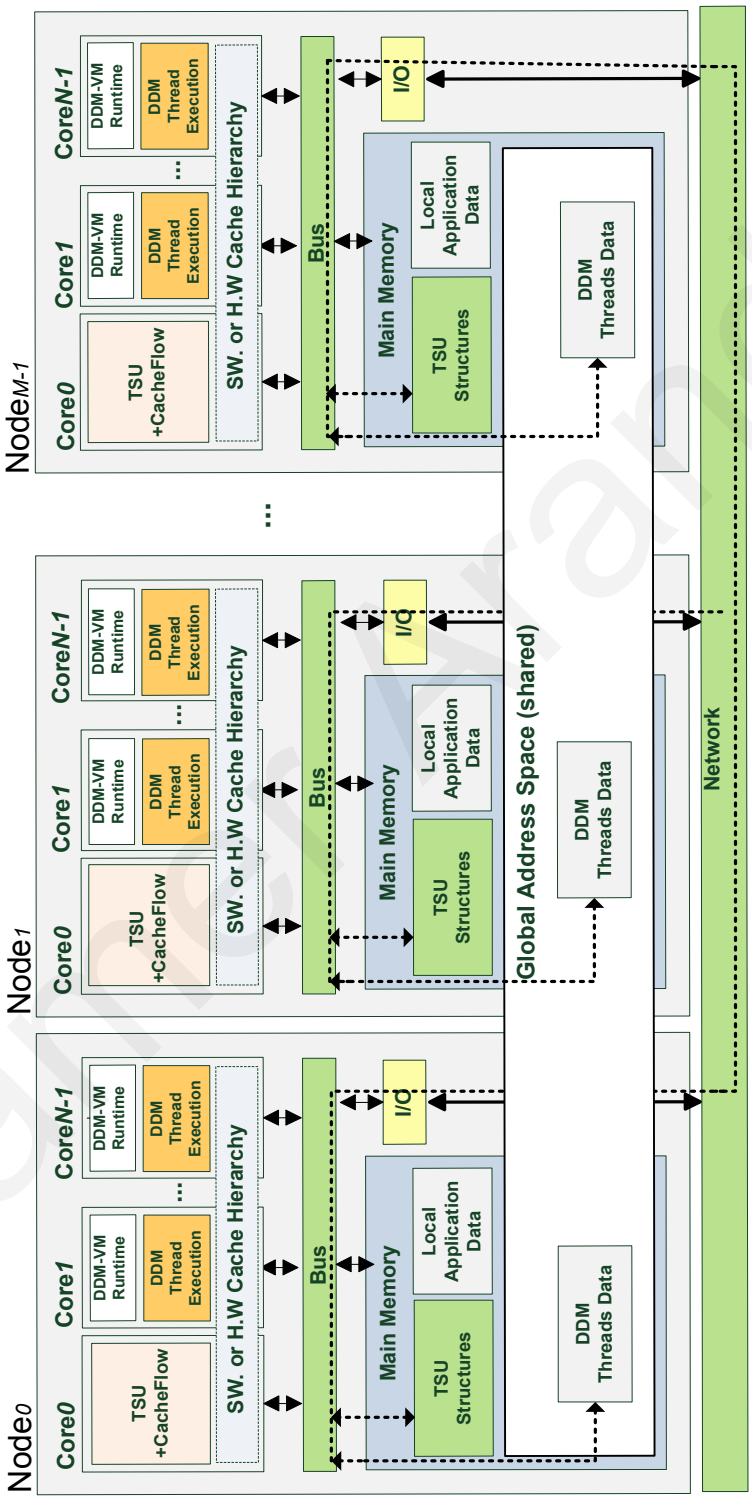


Figure 32: The Distributed DDM-VM Architecture

4.1 Overview

The inherent tolerance for latencies of the DDM model allows extending the execution across multiple distributed nodes with minimal overheads. This is achieved by tolerating inter-node latencies resulting from data and synchronization communications with the execution of threads.

The main difference between single-node and distributed/multi-node DDM execution is the introduction of remote memory accesses resulting from producer and consumer threads running on different nodes. To this end, we employ data *forwarding*, in which the data produced by a thread is *forwarded* to the node where the consumer is scheduled to run. We facilitate this by supporting a Shared Global Address Space across all the nodes. A special TSU module: The Network Interface Unit (NIU) is implemented to handle the low-level communication operations.

In terms of the distribution of threads across the cores of the system nodes, this work explores a *static* scheme, in which the mapping is determined at compile time and does not change during the execution. This simplifies the scheduling and data management tasks and, in the presence of an accurate knowledge of the threads execution loads, can lead to a very efficient and balanced parallel execution. It is important to note that a static distribution only specifies *where* the thread will be scheduled once its ready, however, *when* the thread is ready is decided based on data-availability.

The benefit of this approach extends to programmability, as aside from the distribution of program data in the GAS across the nodes at startup and gathering the results after the program execution, *distributed* DDM-VM programs are fundamentally the same as *single-node* ones.

Next, we highlight the additions and modifications to the TSU structures & operations that are required for supporting distributed execution. Following that we describe the management of

the memory address space and the program data in detail and conclude by re-visiting program termination in the context of distributed execution.

4.2 The Distributed Thread Scheduling Unit (TSU)

The DDM-VM runtime adopts a distributed organization consisting of multiple TSU units (one per node[1]) communicating across the network to coordinate the overall DDM execution. As shown in Figure 32.

4.2.1 The TSU Structures

The Graph Memory (GM) holds the *meta-data* of *all* the program threads on all the nodes. As an optimization, for each node, we only load the *meta-data* of the threads that are expected to execute on that node. The Synchronization Memory (SM) on the other hand requires extra attention as the allocation of SM entries of a thread is directly influenced by the assigned scheduling policy. We discuss this issue in detail in Section 5.6.4. The rest of the TSU structures remain unchanged, however, we added 2 new structures to support distributed execution:

The Distributed Acknowledgement Queue (AQ)

This queue holds decrement RC requests coming from the TSUs on the remote nodes.

Forward Table (FT)

This table holds the address and size of the data that will be forwarded to remote nodes.

[1]node: multi-core processor

4.2.2 The TSU Operations

When the TSU is notified that a thread finished execution, it queries the scheduling policy for the (ThreadId,*context*) of each consumer invocation to get the identifier of the core this invocation is mapped to. If the core is on the same node, an entry is inserted in the AQ of the local node TSU. However, if the core belongs to a remote node, a message containing the invocation (ThreadId,*context*) is sent to the remote node. When the message is received, a request to decrement the RC of that invocation is enqueued in the *distributed* AQ on that node. In addition to the request message, the data produced by the thread is also forwarded to the remote node.

The TSU on each node continuously checks the local AQ(s) and the distributed AQ to decrement the RC of threads invocations. Once the RC of a specific thread invocation reaches zero, its information is moved into the WQ and the rest of the activities proceed as described in 3.3.2.1 & 3.5.1, save the additional steps required for managing the forwarding of produced data to consumers running on remote nodes as will be detailed in Section 4.3.1.

4.2.3 The Network Interface Unit (NIU)

The TSUs on all the nodes communicate and cooperate to perform the various synchronization and data management tasks. To support this communication a new software module is added to the TSU: The Network Interface Unit (NIU). The NIU was used in [73] to support communication amongst distributed single-processor nodes, however it was implemented as a hardware module. In this work the NIU is implemented as a software module that relies on the underlying network hardware interface. Initially, we have considered using MPI [36] for handling the low-level network connectivity in the NIU, however, due to the expected overheads of invoking an external library and our need for customized communication, we have developed our own optimized connectivity layer using non-blocking TCP sockets.

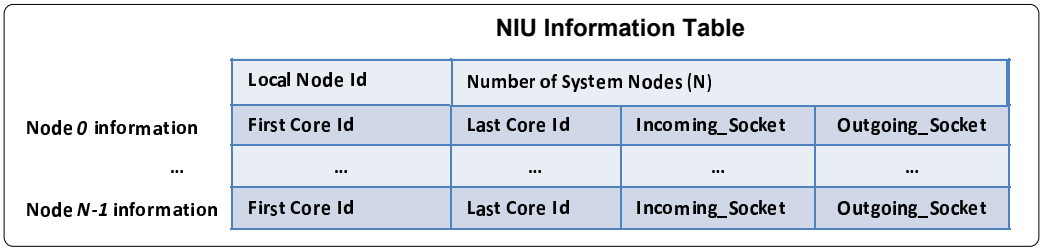


Figure 33: NIU Information Table

The NIU is responsible for managing the network initialization, establishing connections with the other nodes in the system and providing communication services to the TSUs during the execution. The NIU also supports distributing/gathering data across the global address space in the system at startup and post-execution of the DDM-VM program.

Network Initialization

Upon executing a distributed DDM-VM application, a special script is invoked on the *root* node and the name of the DDM-VM executable is passed along with a *peerlist* file as parameters. The *peerlist* file includes the IP addresses of all the nodes in the system. The programmer can also control how many cores to utilize per node in the file. After parsing the *peerlist* file the script copies the DDM-VM executable and *peerlist* file to the nodes and the executable is started on each node. The first task at the application startup is the initialization of the DDM-VM runtime, which invokes the network initialization procedure in the NIU.

In the initialization stage the NIU on each node establishes connections with all the other nodes. For each node two non-blocking sockets are allocated, one for sending (*outgoing* socket) and one for receiving (*incoming* socket). Once the connections are established through the sockets, the NIUs exchange information related on the number of cores utilized for DDM execution on each node. This information is maintained in a table and used later by the TSU to specify on which node each core is located. Figure depicts this table 33.

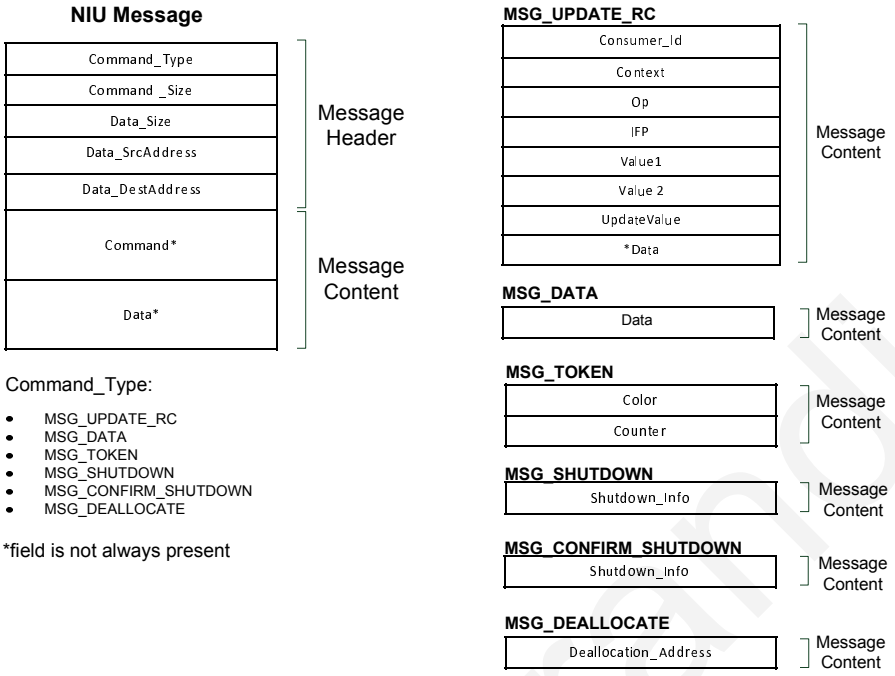


Figure 34: NIU Messages

4.2.3.1 NIU Services

The NIU abstracts the underlying network and provides the TSU with a simple communication interface. The TSU uses the interface to exchange:

- Synchronization commands or messages: the most important one is the request to decrement the *RC* of a specific consumer invocation.
- Data *forwarding*: when a thread produces data that is needed by a consumer on a remote node, the TSU passes the data to the NIU to *forward* it to the remote node.

The NIU is implemented in away that tolerates the latencies of network communication by overlapping its work and data transfers with threads execution and the rest of the TSU work. The NIU module is naturally split into two main independent sub-units:

- The *send* sub-unit: responsible for sending commands and forwarding data to remote nodes. Both commands and data are first encapsulated in a simple message with a header describing the content, before they are sent through the *outgoing* socket associated with the remote node. The sending operation returns when the messages have been stored in the O.S. network layer buffers.

- The *receive* sub-unit: responsible for receiving and processing the messages sent from remote nodes. It continuously polls the *incoming* sockets corresponding to the rest of the nodes in a round-robin fashion. The received messages are processed according to their type. Figure 34 illustrates the format and contents of the different NIU messages. In the case of decrement RC request commands (*MSG_UPDATE_RC*), the information of the message is inserted as an entry in the *distributed* AQ. The rest of the command types are explained in the subsequent sections.

The overlapping of the work of both sub-units in relation to the rest of the tasks in the system depends on the DDM-VM implementation, as will be described next.

4.2.3.2 The Distributed DDM-VM$_c$

In the DDM-VM$_c$ implementation the PPE is the only core that have access to I/O and operating system services. Thus, both sub-units are executed on the PPE, however, the *receive* sub-unit is launched in an *auxiliary* thread running in parallel to the main thread executing the rest of the TSU work. Hence, we take advantage of the fact that the PPE is a SMT processor supporting two hardware threads [63].

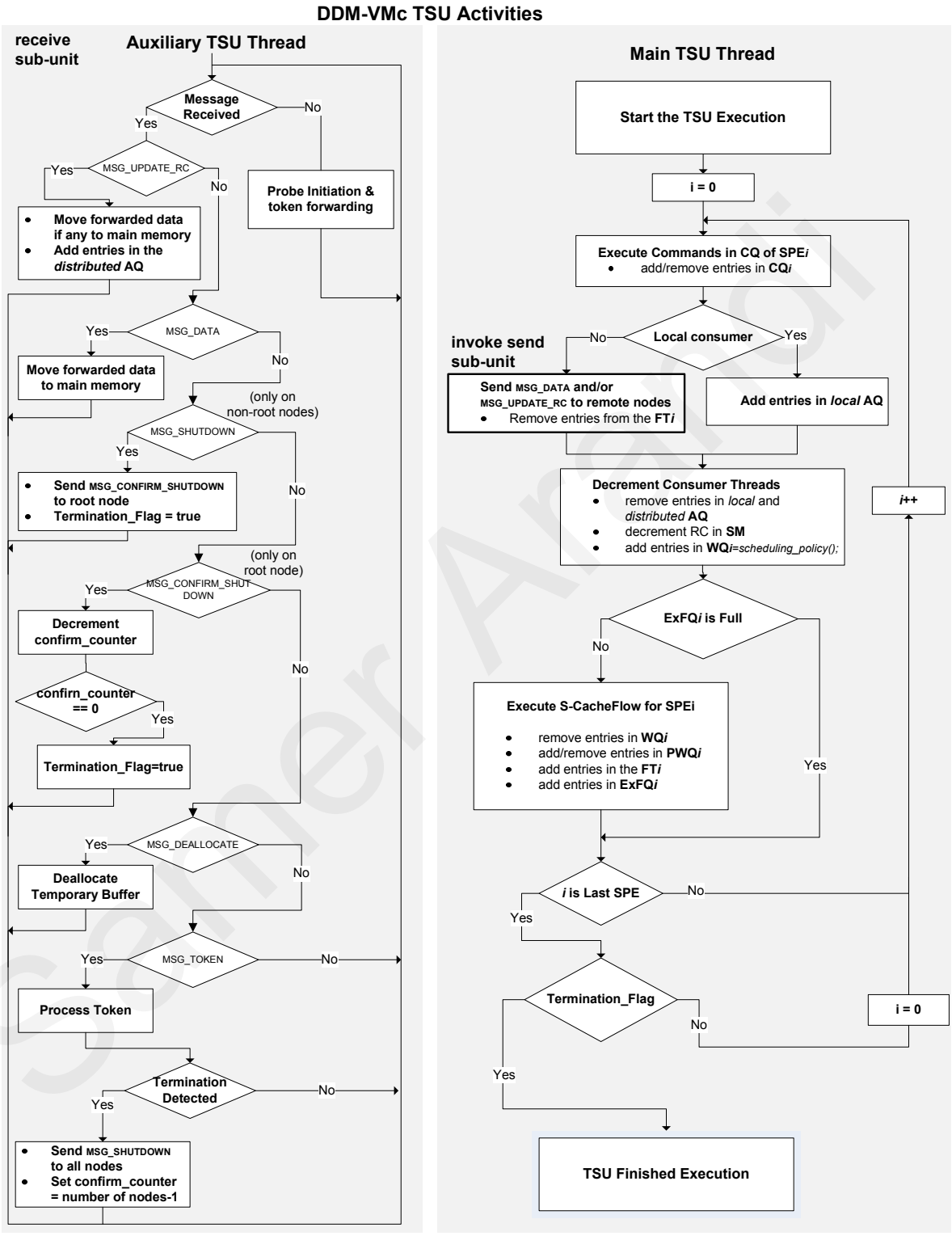


Figure 35: TSU Activities on the PPE - Main and Auxiliary PPE Threads

The *send* sub-unit is directly invoked by the TSU and so it runs in the context of the main thread, while the *receive* sub-unit is concurrently processing incoming messages. This is illustrated in Figure 35. The right side box in the figure shows the activities performed by the main TSU thread, which runs concurrently with the left side box showing the activities of the auxiliary thread. Utilizing this design we overlap the work of the *receive* sub-unit with the work of the TSU and the *send* sub-unit, in addition to overlapping the work of the NIU in general with the execution of the threads on the SPEs.

4.2.3.3 The Distributed DDM-VM$_s$

In the DDM-VM$_s$ implementation the *receive* sub-unit is launched in an auxiliary thread (that is pinned to the same core running the TSU) similar to the DDM-VM$_c$, therefore yielding similar benefits. However, unlike the the DDM-VM$_c$, we further distribute the work of the *send* sub-unit across the cores. This is possible because the services of the *send* sub-unit are invoked by the DDM-VM runtime threads on the cores, which unlike the Cell SPEs, have access to I/O and O.S. system services. This distribution removes the tasks of the *send* sub-unit from the critical path of the TSU. Figure 36 illustrates the main TSU thread, the auxiliary TSU thread running the receive sub-unit and the activities of the runtime threads, which invoke the services of the send sub-unit. Finally, a lock was added on accessing the *outgoing* socket in the *send* sub-unit, since more than one runtime thread might require sending a message to the same destination node simultaneously.

4.3 The Memory Address Space and the Program Data

The DDM-VM supports a Distributed Shared Memory (DSM) [100] abstraction in which part or all of the main memory address space on each node is mapped to the Global Address Space (GAS) of the DSM. An address referring to the GAS consists of the ordered pair **(node_id,local_address)**.

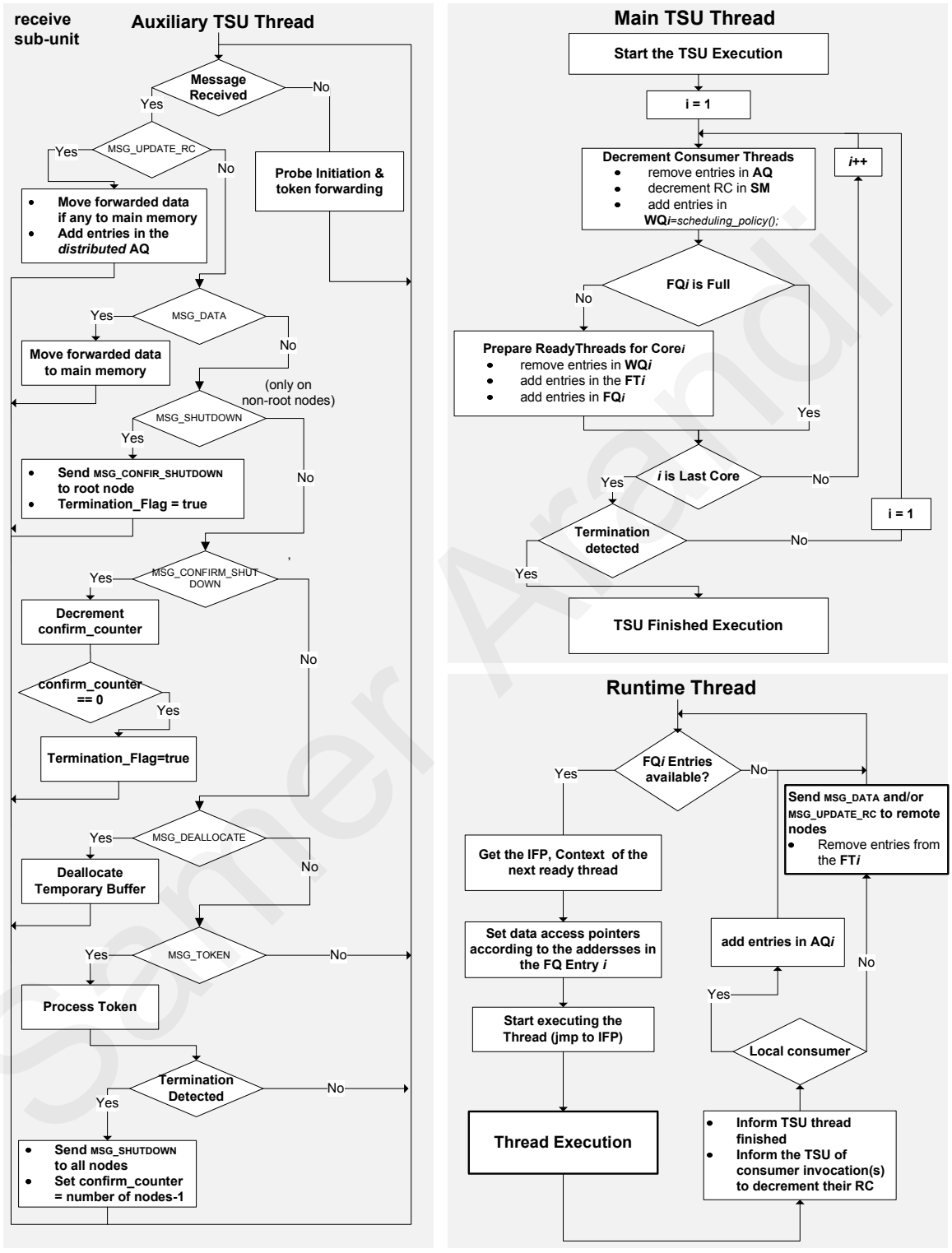


Figure 36: Distributed DDM-VM$_s$ TSU & Runtime Threads Activities on All the Cores in the System

The first component refers to the node identifier and the second refers to a conventional main memory address on that node. Figure 32 illustrates the GAS across the system nodes.

Coherence-management operations typically associated with DSM systems [100] are not required between the nodes, because produced data is *forwarded* to consumers running on remote nodes. Coherence operations are only required within each node's memory hierarchy and so it is either managed by hardware in the DDM-VM$_s$ implementation or by the S-CacheFlow module in the DDM-VM$_c$ implementation.

The mapping of the program data into the GAS depends on the assignment of the program threads. The data of a certain invocations is mapped to the part of the GAS belonging to the node where this invocations is scheduled to run. The movement of data between producers and consumers running on different nodes during the execution is managed automatically by the DDM-VM without the intervention of the programmer.

A number of runtime calls facilitate the allocation and release of the data in the GAS. The calls also abstracts the distribution and gathering of data amongst the nodes. These calls internally invoke the services of the NIU to move the data between the main memories of the nodes.

4.3.1 Data Forwarding and CacheFlow Operations

DDM-VM employs *data forwarding* [99, 68] for tolerating data communication latency across the nodes. The node where a *producer* thread executes *forwards* the produced data to the remote node where a *consumer* thread is scheduled to execute, as soon as the producer thread finishes execution. This increases the chance of hiding the communication latency and eliminates the need for remote read operations, which results in reducing the total communication cost since remote read operations are usually more costly that remote write operations [71]. This also eliminates coherence management operations as previously mentioned.

The forwarding of data completes before decrementing the RC of the consumer thread. Consequently, when a thread RC reaches zero, its data is guaranteed to reside in its node main memory. This allows us to use our intra-node data management techniques without a change except for two additional operations needed to manage the data forwarding, which are described next.

Data Forwarding - Preparation

The first operation is introduced upon moving a thread entry from the WQ to the FQ. The address of each of the thread output data (produced data) is examined. If the address refers to a memory location on the local node then the thread is processed as before without a change. If the address refers to a location on a remote node (which indicates that the thread has a consumer on that node), a temporary buffers is allocated to hold the data and an entry is added into the Forward Table (FT). The following steps describe this procedure in detail:

- A temporary buffer is allocated on the local node with a size equal to the size of the output data
- The output data address is replaced by the temporary buffer address
- A *data forward* entry is added to the FT in the TSU. The entry contains: the address of the temporary buffer (the source of the data forward), the original output data address on the remote node (the destination of the data forward) and the size of the forwarded data.

After that the execution of the TSU activities (including the S-CacheFlow tasks on the DDM-VM$_c$) proceed as before without a change. It is important to note that after the thread finishes execution the produced data exists in the the temporary buffer.

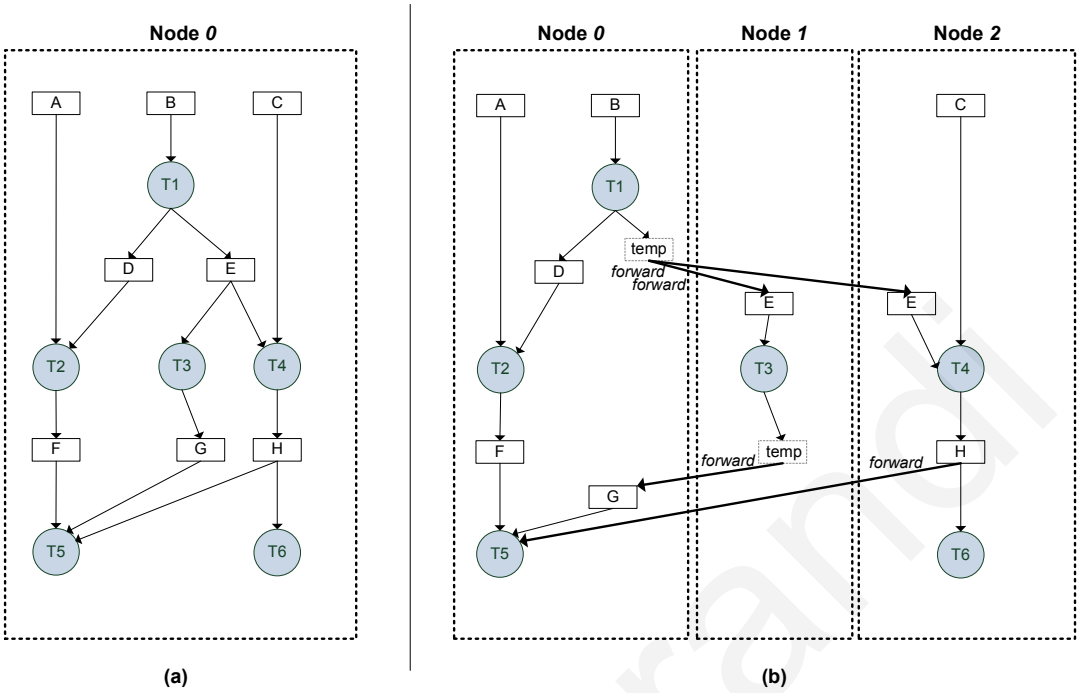


Figure 37: Data Forwarding Example

Data Forwarding - Transfer

The second data forwarding operation is introduced after the thread finishes execution. For every remote consumer of the thread, the TSU checks the FT for data forward entries heading to the same remote node. Once found, the forwarded data and the decrement RC request messages are then sent via the NIU services as explained in 4.2.3.1. To reduce the number of messages and efficiently utilize the network bandwidth we bundle the forwarded data with the decrement RC request into one message. When this *compound* message is received at the remote node, the data is copied to the main memory first and then the decrement RC request is inserted in the *distributed* AQ. In addition, a message (MSG_DEALLOCATE) is sent back to the source node to release the allocated temporary buffer.

Multiple Remote Consumers

The DDM-VM supports threads that produce data consumed by multiple remote consumers running on different nodes. In this case a *list* of output addresses is provided (instead of only one) and the DDM-VM *forwards* the data to all the locations in the list by creating an FT entry for each remote address. Note that in this case one temporary buffer is allocated and used as the source of all the data forwards. If any of the addresses in the list is located on the same node (one of the consumers is a local one), the TSU uses this address as the source of the data forwards and no temporary buffer is allocated.

Data Forwarding Example

Figure 37 shows an example of a simple DDM-VM program comprising 6 DDM threads. In part *(a)* the program is running on a single node and so no data forwarding is required as all the data is allocated in the main memory of the same node. In part *(b)* of the figure the program threads are mapped across 3 nodes. Data items E, G and H produced by threads T1, T3 and T4 respectively, are consumed by threads running on remote nodes and so must be forwarded once the producing threads finish execution. The data forwards are shown in thick arrows and the temporary buffers allocated to hold the produced data are shown in dashed boxes. Note that in the case of data item H, because it is also consumed by one thread on the same node, no temporary buffer is allocated.

4.4 Distributed Execution Termination

The two termination detection approaches described in 3.3.5 need to be revisited in the context of distributed DDM execution.

4.4.1 Explicit Termination Approach

The first *explicit* approach that designates a specific thread invocation as the *last* executed one is still valid, albeit with a simple extension. The TSU that is notified that the *last* thread invocation finished execution informs the root node, which broadcast a *termination* message (MSG_SHUTDOWN) to the rest of the TSUs. The TSUs respond with a shutdown confirmation message (MSG_CONFIRM_SHUTDOWN) to the root and so a graceful complete system termination is achieved.

4.4.2 Implicit Termination Approach

The second *implicit* approach, however, requires additional work to adapt for distributed execution given the lack of complete knowledge of the global state. Detecting termination in this manner falls under the *distributed termination problem* [42], which is one of the fundamental problems of distributed execution. Numerous solution have been proposed for this problem [81]. We select a solution based on the algorithm proposed by Dijkstra and described in detail in [106], as it requires minimal message exchange and can be implemented as an extension of the second approach.

The algorithm assumes termination when: *the state of all the nodes is passive (idle) and no messages are on their way in the system*. The purpose of the algorithm is to enable one of the nodes, say *node 0* (the root), to detect that this state has been arrived at. The algorithm is described as follows:

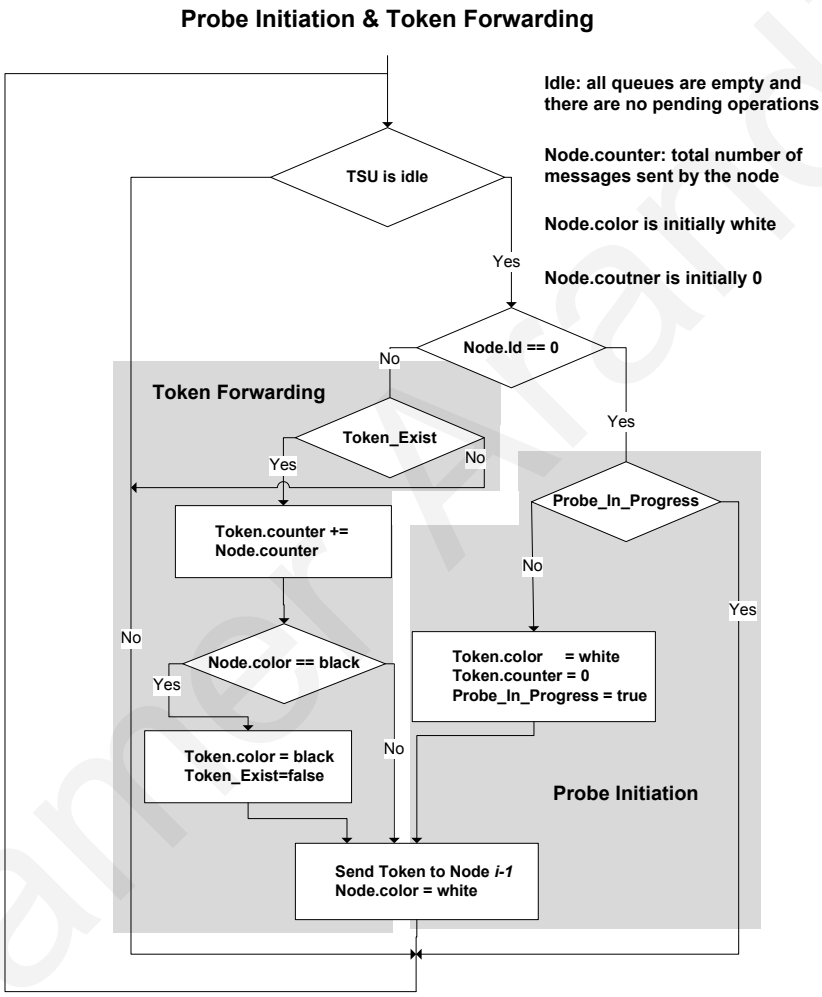


Figure 38: Distributed Termination Detection - Probe Initiation and Token Forwarding

The Algorithm:

- Every node maintains a message counter *c*. The counter is incremented when a message is sent and decremented when a message is received. The total sum of the counters on all the nodes equals the number of messages pending in the network.
- When *node 0* initiates a detection probe, it sends a token with a value *0* to *node N-1*. Every *node i* keeps the token until it becomes passive; it then forwards the token to *node i-1* increasing the token value by *c*.
- A color is assigned to each node and token (initially all are white). When a node receives a message, the node turns *black*. When a node forwards the token, the node turns *white*. If a *black* node forwards the token, the token turns *black*, otherwise the token keeps its color.
- When *node 0* receives the token again, termination is concluded if: (i) *node 0* is passive and *white*, (ii) the token is white, and (iii) the sum of the token value and c is *0*. Otherwise, *node 0* may start a new probe.

In our implementation the state *passive* refers to the state when all the TSU queues are empty and there exists no pending in-flight operation. Moreover, initiating a probe occurs whenever the root node becomes *passive* and no previously initiated probe is in progress. Once termination is detected on the root node a termination message is broadcast to the rest of the TSUs to achieve a graceful termination as described in the first approach. Sending and receiving token messages are performed via the NIU services.

Figure 38 and 39 depict the two parts of the implemented algorithm. Both parts are executed by the the Auxiliary TSU thread. The first figure illustrates the part executed at every cycle of the

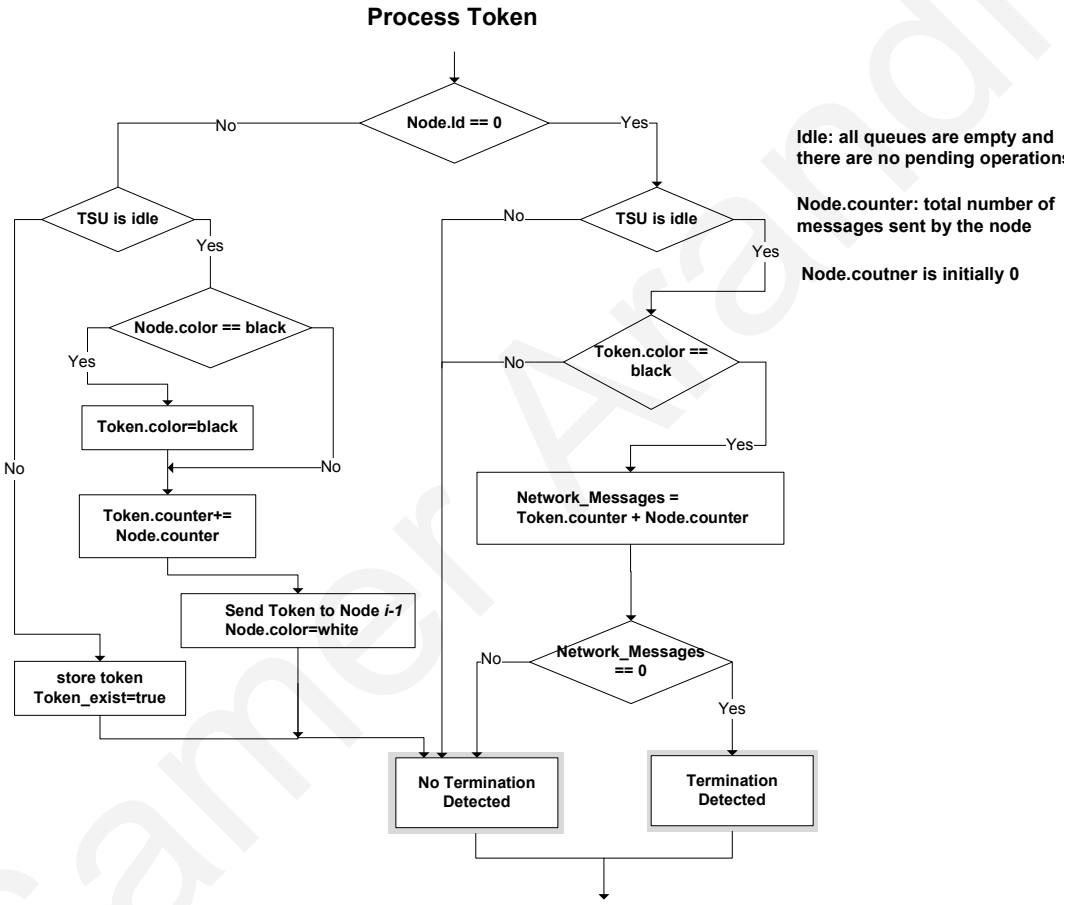


Figure 39: Distributed Termination Detection - Token Processing

auxiliary thread and the second illustrates the part executed when a token is received from another node.

Chapter 5

Programming Methodology and Optimizations

5.1 Introduction

In this chapter we present the programming methodology and tool-chain utilized with the DDM-VM and provide a number of programming examples. We also propose a technique for handling programs containing dependencies that cannot be uncovered at compile-time. Further, we present a number of optimizations employed to improve the performance of the DDM-VM. We also describe the support for distributed DDM execution and conclude the chapter by demonstrating the monitoring and visualization tools. Part of the work in this chapter has been presented in [9].

5.2 Dynamic Data-flow

The Data-Flow execution model [34, 13, 130] relies on the availability of input data for the execution of each operation (as opposed to a program counter in the Control-flow model). This allows it to tolerate memory and synchronization latencies and provide a distributed concurrency control mechanism. Moreover, the side-effect free semantics of this model allows it to expose

all the parallelism in a program, as it enforces partial ordering that corresponds to the true data dependencies.

Dynamic data-flow architectures [13, 11, 130] (as opposed to static data-flow ones) allow loop iterations and subprogram invocations to proceed in parallel via the *tagging* of data tokens.

The DDM-VM utilizes the distributed synchronization mechanism of dynamic Data-Flow as described by the U-Interpreter [11]. Each data value is associated with a unique tag; a *Token* V[c,s,i] is made up of the value V and the tag [c,s,i], "c" is the context identifier, "s" is the destination address and "i" is the iteration identifier. The U-interpreter provides a formal distributed mechanism for the generation and management of the tags at execution time.

We briefly explain the basic principles of the U-Interpreter with the aid of the inner product example shown in Figure 40-a, the corresponding dynamic Data-Flow graph is shown in Figure 40-b. The [L] operator in Figure 40-b creates a new loop *context* by adding a loop identifier to the *tag* and initializing it to zero. Every time a token goes around in the loop the [D] operator increments the iteration part of the *tag*. T1 part of Figure 40-a demonstrates the process of *tag* creation for the first iteration of the loop. Note that in dynamic Data-Flow an actor can fire only when inputs with identical *tags* are available at all its input ports. The whole process is repeated until the loop predicate evaluates to false, the switch actor then routes the last token to the [D^{-1}] operator which restores the iteration part of the *tag* to zero and then the [L^{-1}] operator remove the loop identifier added by the [L] operator thus restoring the *tag* it had before entering the loop.

5.3 Data-Driven Multithreading (DDM)

In the DDM model, the tagged token matching operations are reduced into virtual memory translations and implemented as updates to the RC entries in the Synchronization Memory (SM)

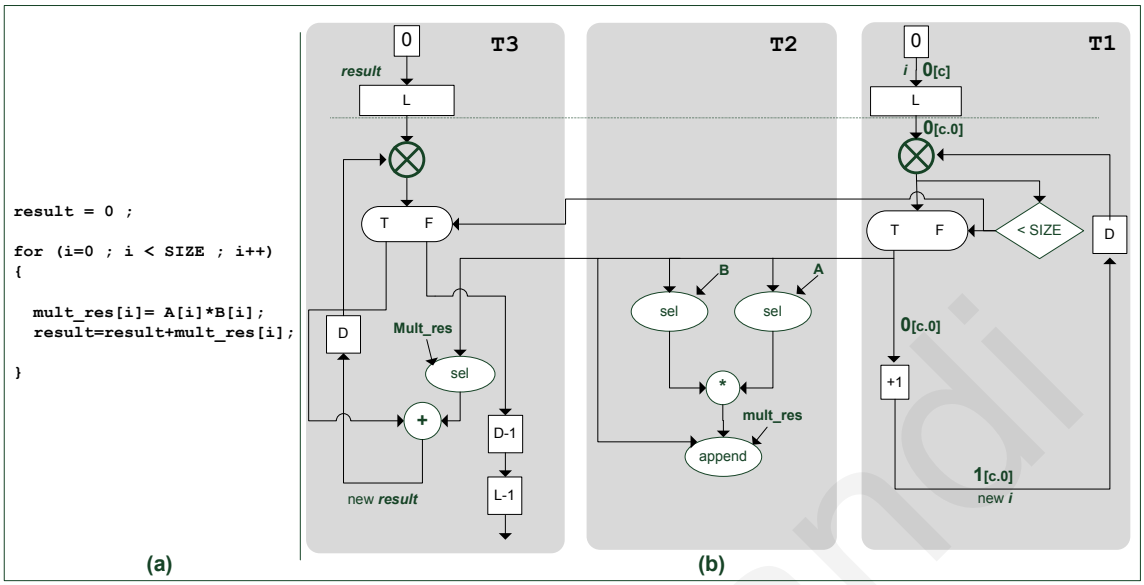


Figure 40: The Vector Dot Product (a) Original Program (b) U-Interpreter Dynamic Data-Flow Graph

structure allocated in main memory. The tag manipulation operators are implemented by the runtime.

A DDM program is represented as a number of re-entrant, inter-dependent DDM threads, along with each thread's *meta-data* (or *synchronization template* in DDM terminology). The threads are identified by the *ThreadId* and *Context*. The *context* corresponds to the tag of the U-Interpreter and uniquely identifies dynamic invocations of each thread. The *context* records the effect of re-entrance and so when mapping loops into DDM threads we derive the value of the *context* from the nested loop indices.

All structures are allocated and mapped to the virtual memory space. The dynamic *context* is combined with static meta-data to uniquely identify each thread Ready Count (RC) entry and its input and output data. Figure 41 shows an example of how the context value of a specific instance of thread 1 (instance with context = 15) is used to access its RC entry in the Synchronization Memory. Moreover, the base addresses of the two arrays accessed by thread 1 (A and

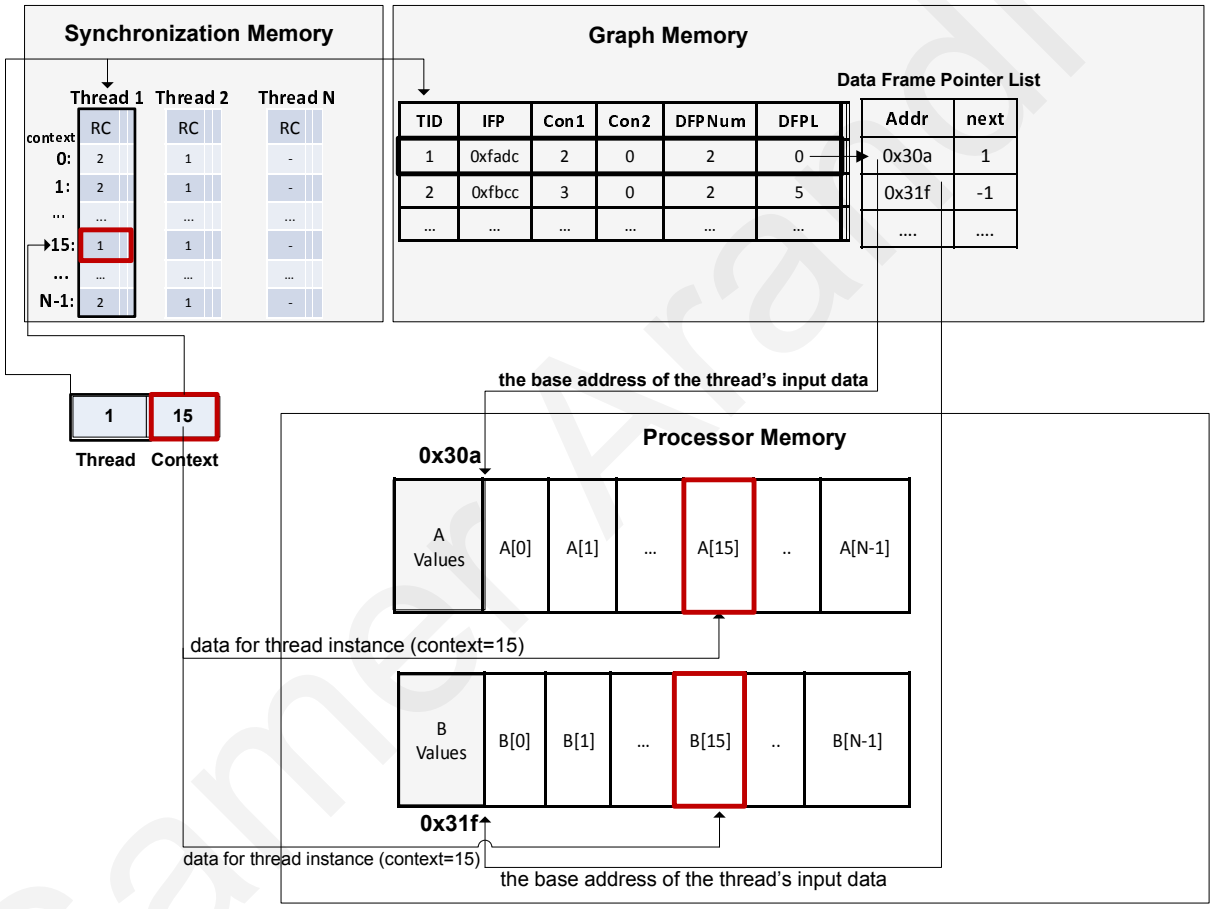


Figure 41: Accessing Thread structures using a combination of the meta-data and the dynamic context in DDM

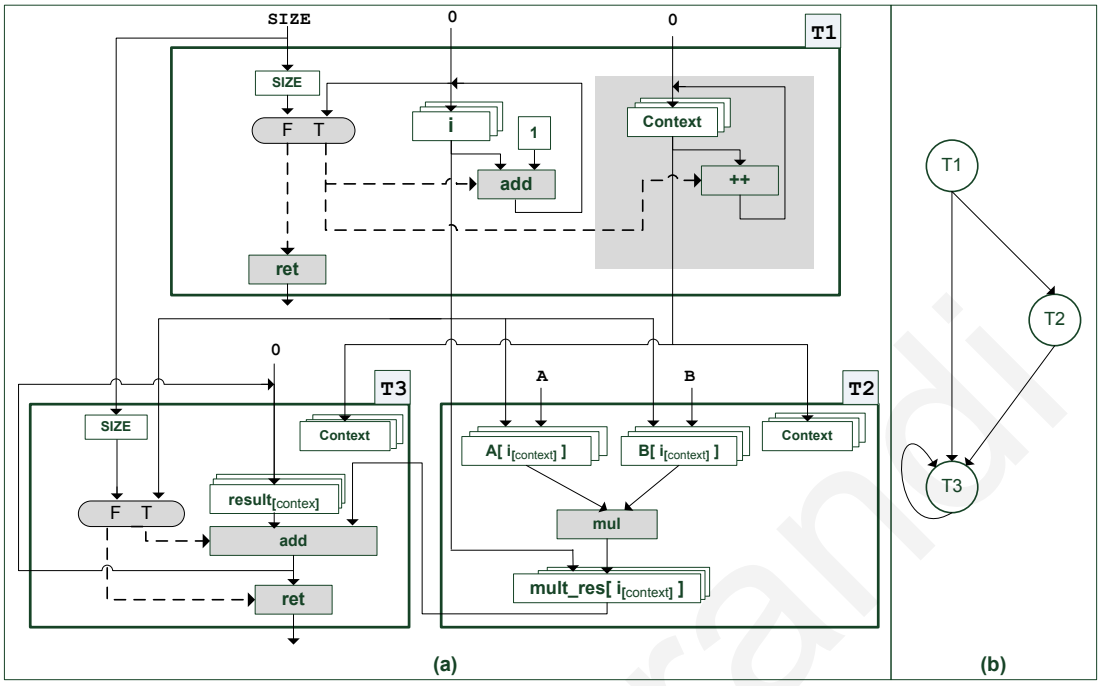


Figure 42: The Vector Dot Product DDM Dependency Graph (a) Detailed view (b) High-level view

B) are retrieved from the GM and combined with the context to access the values for the specific instantiation instance.

Vector Dot-Product Example

Back to the earlier vector dot product program, Figure 42 depicts the equivalent DDM graph that is composed of three threads T1, T2 and T3. The outer loop is implemented by DDM thread T1 and the two operations in the body of the loop are implemented by threads T2 and T3, respectively. Figure 42-a illustrates a detailed view of the graph. Solid arrows represent data dependencies and actual movement of data instances between the threads. Dotted arrows represent only data dependencies amongst the threads. Shaded rectangles represent operations, non-shaded single rectangles represent static values and non-shaded multiple rectangles represent dynamic values.

Thread T1 generates the values of the loop index *i*. The shaded portion of T1 generates the *context* value and corresponds to the [D] operator of the U-interpreter of Figure 40-b. The reader will notice that in this example the *context* value is identical to the loop index value. In such cases we can omit the extra graph that generates the *context* and use the loop index for both. However, in the general case we need to generate the *context* values that correspond to each invocation of the loop and in recursive constructs. As mentioned previously, the *context* value is used by the program threads to access the data value(s) corresponding to each invocation.

Thread T2 performs the multiplication and uses the value of *i* as an index to the input and output vectors. T2 then notifies the TSU to update the *readycount* of the invocation of thread T3 consuming the produced value. Thread T3 accumulates the multiplication of the two vectors in *result* and informs the TSU to update the readycount of the next invocation of T3. When the loop predicate evaluates to false *result* contains the vector dot product value.

5.4 DDM-VM Programming Methodology

The success of any alternative execution model depends on many factors, but the foremost are the programmability and the efficiency of the resulting programs. To this end, we have a number of alternative approaches for programming the DDM-VM:

1. *Macro-based*: this is a low-level interface for programming the virtual machine that utilizes a set of C macros. This interface serves as the target for the rest of the approaches. It permits the expert programmer total control of the DDM-VM program.

2. *T-Flux preprocessor*: utilizes the TFlux directives and preprocessor tool originally developed in [113]. A subset of the directives is extended to generate the DDM-VM macros. This work is a collaborative effort with the TFlux team.

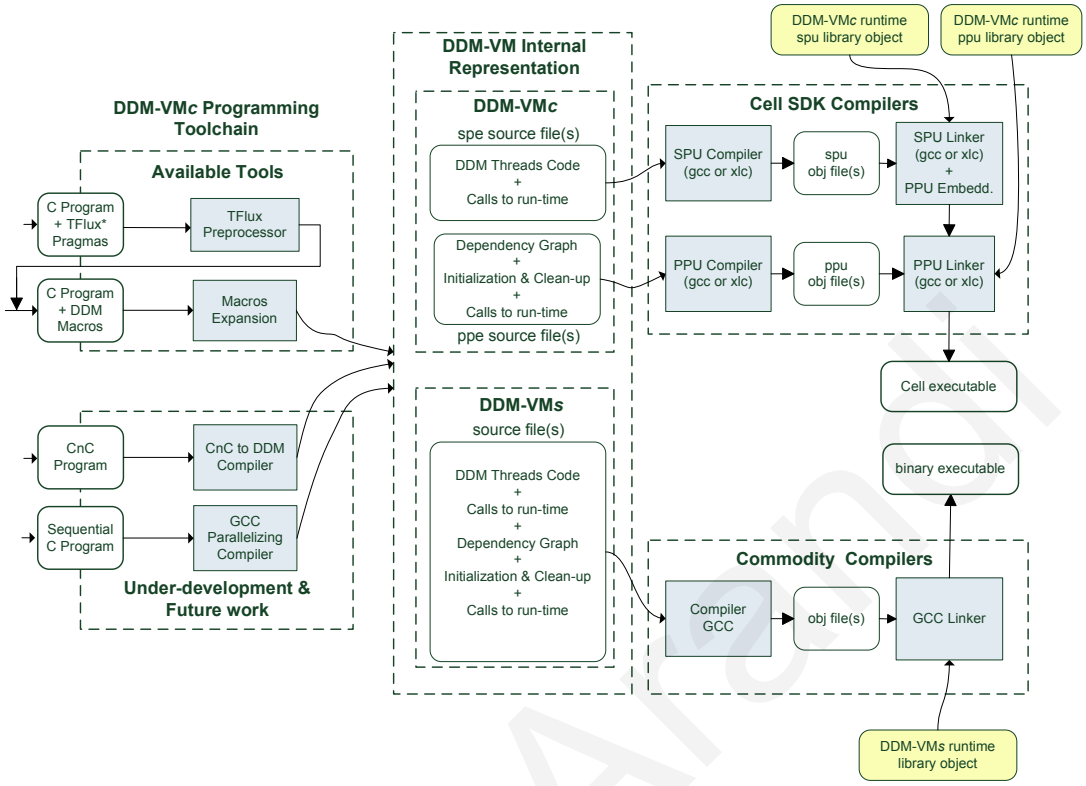


Figure 43: The DDM-VM$_c$ Programming Toolchain

3. *GCC-based auto-parallelizing compiler*: utilizes the GCC compiler to automatically generate code targeting the DDM-VM. This project is a collaboration effort that is still under development with encouraging preliminary results.

4. *CnC-to-DDM compiler*: utilizes the CnC [22, 23] declarative parallel programming language to generate the DDM-VM macros with the help of a compilation tool. We describe this approach in detail in the Future Work Chapter.

We explain the programming in this chapter using the macro-based approach, as it is the one we used to develop the applications and is the most detailed interface, however, we expect the programmers to use the other approaches for writing DDM-VM programs.

The resulting code of the DDM-VM program, which is generated by any of the approaches is compiled using the target platform compilers and linked with the DDM-VM runtime. In the

case of the DDM-VM$_c$ the Cell SDK compilers are used. In the case of the DDM-VM$_s$ the GCC compiler of the underlying architecture is used. Figure 43 shows an overview of the DDM-VM toolchain.

5.4.1 The Low-Level Interface: DDM-VM Macros

DDM-VM represents DDM programs using a set of C *macros* that expand into calls to the DDM-VM runtime. The macros are described in Table 1 and perform the following tasks:

- Identify thread boundaries
- Create the dependency graph of the threads
- Manage the dependency graph & the updating of consumers

The process of mapping a program using the DDM-VM macros is explained with the aid of the blocked matrix multiplication. The code of the original program is depicted in Figure 44-a. We chose the width of matrix B to be equal to one block for simplification.

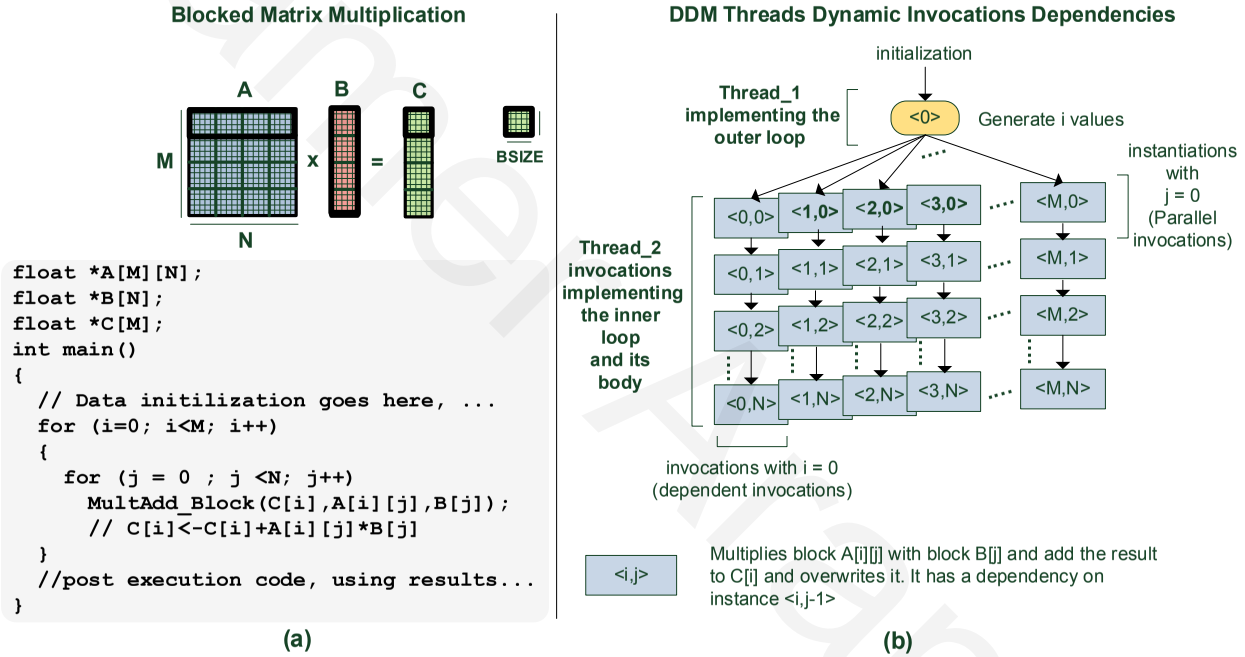


Figure 44: The Blocked Matrix Multiplication Application (a) The original code of the application (b) Dependencies across the dynamic invocations of the DDM threads

5.4.2 Identifying the Boundaries of DDM Threads

Figure 44-b depicts one possible mapping of this application to a DDM-VM program. The outer *for* loop is mapped into a DDM thread called *THREAD_1* and the inner loop and its body are mapped into a DDM thread called *THREAD_2*. We execute the multiplications in blocks, two blocks multiplied at a time, since in the case of the DDM-VM$_c$ the limited space of the Local Store (LS) memory precludes us from executing the entire row times the column. In the case of the DDM-VM$_s$, we are free to use this program organization or opt for one that multiplies the entire row with the entire column since the memory limitation doesn't exist. Each instance of *THREAD_2* calls the *MultAdd_Block* kernel that performs the operation *C[i]=C[i]+A[i][j]*B[j]*. This multiplication-accumulation introduces a dependency between the successive iterations of the inner loop. Thus, no parallelism can be exploited by unraveling this loop. For that reason the index generation of the inner loop and its body are merged into a single DDM thread (*THREAD_2*). This is an optimization that avoids the overhead of having a thread that generates the loop indices for a sequential loop.

Figure 44-b illustrates the dynamic invocations of the DDM threads with their dependencies represented as arrows. Each invocation is labelled with its *context*. For example since *THREAD_2* invocations represent two-level nested loop iterations, the *context* of each invocation is composed of the two loop indicies $<i,j>$.

The corresponding code of the DDM-VM program threads is shown in Figure 45. The macros *DVM_THREAD_START* and *DVM_THREAD_END* mark the boundary of the threads. The former macro also assigns the unique thread identifier (ThreadId) and the latter informs the TSU that

```
DVM_THREAD_START(TID_THREAD_1);

for (i = 0 ; i < M ; i++)
  DVM_UPDATE(CONS1,OP_SET_CONTEXT,MAKE_CONTEXT_D(i,0));

DVM_THREAD_END();

DVM_THREAD_START(TID_THREAD_2);
DVM_LOOKUP(float *,A);
DVM_LOOKUP(float *,B);
DVM_LOOKUP(float *,C);
GET_CONTEXT_D(DVM_CONTEXT,i,j);

MultAdd_Block(C,A,B);

if (j < N-1)
   DVM_UPDATE(CONS1,OP_SET_CONTEXT,MAKE_CONTEXT_D(i,j+1));
DVM_THREAD_END();
```

Figure 45: The code for the DDM threads using the DDM-VM macros

the thread finished execution. The *DVM_CONTEXT* is a variable set by the runtime to the current value of the *context*. The *GET_CONTEXT* macro is used to retrieve the *context* components corresponding to the values of the two loop indices *i* and *j*.

The *DVM_LOOKUP(TYPE,VAR)* macro is used to retrieve the address of one data input/output of the thread. It stores the address in the pointer variable *VAR* after type-casting it to *TYPE*. In the case of the DDM-VM$_c$ using this macro is required as it gets the address of the data in the LS. When running the application on the DDM-VM$_s$, however, this macros is optional (as the data can be accessed directly in the main memory space), except in the case of threads accessing dynamically allocated data, which can occur in distributed DDM execution. In general, it is advised to always use the *DVM_LOOKUP* macros so as to unify DDM-VM programs regardless of the underlying architecture.

5.4.3 DDM Dependency Graph and Context Maintenance

The *DVM_UPDATE* macro informs the TSU to decrement the *readycount* of the thread consumers. It specifies the dynamic invocation of the consumer to update by designating the *context*

value of that invocation using the *MAKE_CONTEXT* macro in combination with a number of operators. The last two macros are used to implement the U-Interpreter *context* operators.

The for loop in the body of *THREAD_1* implements the [D] operator by incrementing the loop index *i*. For every value of *i*, the thread invokes the *DVM_UPDATE* macro and passes the OP_SET_CONTEXT operator as a parameter which implements the U-Interpreter [L] operator. This operator creates the *context* of the invocations of *THREAD_2* by informing the TSU to update the *readycount* of the first invocation of every group of the dependent invocations of *THREAD_2*. Each invocation of *THREAD_2* multiplies one block from A with one block from B and adds the result to one block from C before storing the final result in that block. The *DVM_UPDATE* macro at the end informs the TSU to decrement the RC of its consumer, i.e., the next dependant invocation of *THREAD_2*. Note that the *DVM_UPDATE* macros correspond to the dependency arrows depicted in Figure 44-b.

5.4.4 DDM Dependency Graph Creation and Execution

Table 1: The DDM-VM Macros

DDM-VM macros	Operation
DDM Thread Boundaries	
DVM_THREAD_START(THREAD_ID)	identifies the first instruction of the thread and assigns the thread identifier
DVM_THREAD_END()	• informs the TSU that the thread has finished execution • relinquishes control to the runtime to execute the next ready thread
DVM_LOOKUP(TYPE,NAME)	retrieves the address of one input/output of the thread. In the case of the DDM-VM*c* it performs DDM Cache Lookup to get the address of the data in the LS
DDM Dependence Graph Creation	
DVM_SET_THREAD_TEMPLATE(THREAD_ID, CONS1,CONS2, READY_COUNT, NUMBER_OF_DFPs, SCHED_MODE,SCHED_VALUE, ARITY, SM_METHOD,SM_VALUE)	creates and loads the thread synchronization template consisting of: the thread identifier, the consumers (if the thread has more than two consumers CONS1 = 0 and CONS2 is a pointer to a list of consumers), the *readycount* value, the number of DFPs, the scheduling policy for the thread, the *arity* which indicates the loop nesting level for the thread and the Synchronization Memory (SM) method to use.
CacheFlow	
DVM_START_DFPL(THREAD_ID**)** **DVM_SET_DFP(**ADDR,SIZE,FLAG**)** **DVM_END_DFPL()**	assigns the information of the DFPL: the address and size of the input/output data of the thread. The flag field specifies the direction of data access (in, out or inout) and the re-use flag when exploiting locality.
DVM_SET_REFCOUNT()	assigns the reference-counter value for data items that will be re-used when exploiting locality
DVM_CACHEFLOW_DFPL_START() **DVM_CACHEFLOW_DFPL_END()**	the two macros declare the start and end of the DFPL helper function called by the TSU, respectively. All the DFP macros must be defined between the two macros.
DDM Dependence Graph Maintenance and Context Management	
DVM_UPDATE(CONS,OP,VALUE)	• informs the TSU which specific invocation of a consumer to decrement its *readycount* when the thread finishes execution • implements the U-Interpreter *context* manipulation operators • in the case of the DDM-VM*c* any produced data is exported to main memory before executing the update
DVM_UPDATE_MULTIPLE(CONS,VALUE1,VALUE2)	informs the TSU to decrement the *readycount* of multiple invocations of a consumer thread.
DVM_UPDATE_THREAD(THREAD_ID,VALUE) **DVM_UPDATE_THREAD_MULTIPLE**(CONS,VALUE1,VALUE2)	the two macros inform the TSU to decrement the RC of one specific thread invocation or multiple ones, respectively. The macros are used with threads that consume initialized data.
DVM_CONTEXT	a variable set by the runtime to provide access to the current value of the thread *context*.
MAKE_CONTEXT(…)/GET_CONTEXT(…)	create/retrieve the value of the context. The value could have single, double or triple *arity*
Thread Scheduling Policy	
DVM_START_SCHEDULE(THREAD_ID**)** **DVM_SET_SCHEDULE(**CoreID**)** **DVM_END_SCHEDULE()**	overrides the default scheduling policy and controls to which core the invocations of the thread are scheduled.
DVM_SCHEDULING_POLICY_START() **DVM_SCHEDULING_POLICY_END()**	the two macros declare the start and end of the scheduling helper function called by the TSU, respectively. All the scheduling policy macros must be defined between the two macros.
Execution	
DVM_EXECUTE()	Starts the execution of the TSU and the scheduling of threads to execution units. It returns after all the loaded DDM threads have finished execution

```
float *A[M][N];
float *B[N];
float *C[M];
int main()
{
  // Data initilization goes here
  // THREAD_1 DDM thread template
  DVM_CREATE_THREAD_TEMPLATE(TID_THREAD_1,    //TID/IFP
                             TID_THREAD_2,0,    //consumers
                             1,                 //readycount=1
                             0,                 //DFPNum=0
                             DVM_DYNAMIC,0,     //scheduling Mode
                             DVM_ARITY_0,       //nesting-level=0
                             DVM_ASSOCIATIVE,0);//Associative SM

  // THREAD_2 DDM thread template
  DVM_CREATE_THREAD_TEMPLATE(TID_THREAD_2,    //TID/IFP
                             TID_THREAD_2,0,    //consumers
                             1,                 //readycount = 1
                             3,                 //DFPNum = 3
                             DVM_CUSTOM,0,      //Scheduling Mode
                             DVM_ARITY_2,       //nesting-level=2
                             DVM_ASSOCIATIVE,0);//Associative SM

    DVM_UPDATE_THREAD(TID_THREAD_1,0); // update the
                  // readycount of thread THREAD_1 (becomes zero)
     DVM_EXECUTE(); // start the execution of the TSU, only
                    //comes back after all threads finish
}
```

Figure 46: Initialization, graph creation, graph execution and post-execution code

Figure 46 depicts the code that runs before and after the execution of the DDM threads in the program. After data initialization, the programmer uses the *DVM_CREATE_THREAD_TEMPLATE* macro to load the *synchronization template* of each DDM thread into the TSU. The template is described in detail in Section 3.3.2.

The programmer then uses the *DVM_UPDATE_THREAD* macro (which is a variant of *DVM_UPDATE* that informs the TSU to decrement the RC of a specific thread directly) to decrement the RC of *THREAD_1* making it ready for execution, before calling the *DVM_EXECUTE* macro which starts the execution of the TSU and the scheduling of threads to the cores. This macro returns only when termination is detected, after which the results can be accessed for processing or storing into files, etc. Figure 47 illustrates the general flow of a DDM-VM program.

The Threads Data

One of the parameters of the the *DVM_CREATE_THREAD_TEMPLATE* macro sets the number of entries in the Data Frame Pointer List (DFPL) of the thread. However, the information of

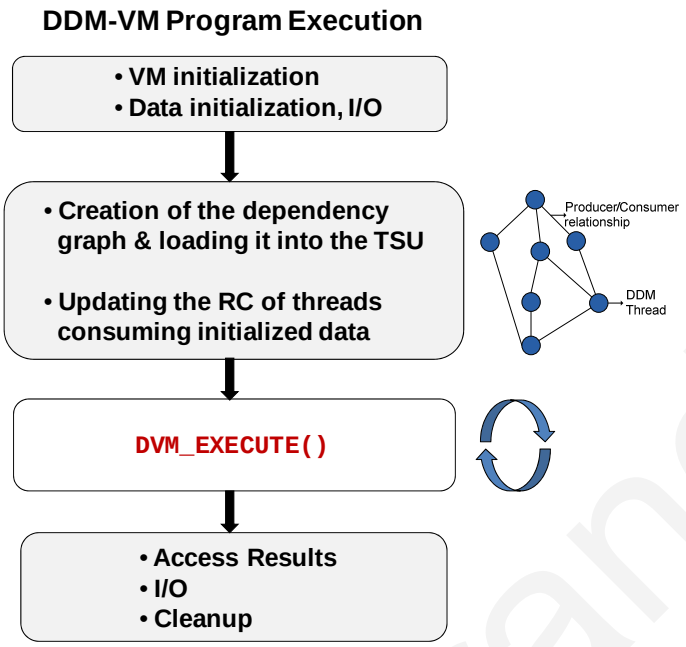


Figure 47: The Flow of a DDM-VM Program Execution

```
// DFPL definitions
DVM_CACHEFLOW_DFPL_START();//start of function called by TSU
int i,j;
DVM_START_DFPL(TID_THREAD_2); // start of DFPL definition
  GET_CONTEXT_D(DVM_CONTEXT,i,j);
  DVM_SET_DFP(A[i][j],BSIZE*BSIZE*4,DATA_IN);
  DVM_SET_DFP(B[j],BSIZE*BSIZE*4,DATA_IN);
  DVM_SET_DFP(C[i],BSIZE*BSIZE*4,DATA_IN|DATA_OUT);
DVM_END_DFPL();        // end of DFPL definition
DVM_CACHEFLOW_DFPL_END(); // end of function called by TSU
```

Figure 48: DFPL definition macros

the DFPL itself is encoded by another set of macros shown in Figure 48. The macros specify the *address*, *size* and *flags* for the data of the thread. The *flags* indicate if the data is accessed for read (DATA_IN), write (DATA_OUT) or both (DATA_IN|DATA_OUT). The *DVM_SET_REFCOUNT* macro is used for assigning the reference-count values when utilizing data locality. The information of the data is directly extracted from the original code. The only difference is that the loop indices used to index the data arrays are replaced by the corresponding components of the *context*. The information of the DFPL is always required in the case of the DDM-VM$_c$. In the case of the DDM-VM$_s$ it is required for supporting data *forwarding* in distributed execution as will be demonstrated in Section 5.5.

```
// Optional Scheduling policy definition
DVM_SCHEDULING_POLICY_START();//start of function called by TSU
int i,j;
DVM_START_SCHEDULE(TID_THREAD_2);
  GET_CONTEXT_D(DVM_CONTEXT,i,j);
  DVM_SET_SCHEDULE(i%NUMBER_OF_CORES);
DVM_END_SCHEDULE();
DVM_SCHEDULING_POLICY_END(); //end of function called by TSU
```

Figure 49: Scheduling policy definition macros

Scheduling Policy

Assigning one of the scheduling policies (described in 3.3.4) to a thread is done via the 6th and 7th parameters of the *DVM_CREATE_THREAD_TEMPLATE* macro. For example to select a dynamic scheduling policy the value *DVM_DYNAMIC* is passed as a parameter to the macro. If a custom scheduling policy is to be implemented the programmer passes the *DVM_CUSTOM* instead and encodes the policy using a number of macros which specify the Identifier of the core to which a thread is scheduled. Figure 49 illustrates an example of a scheduling policy that assigns invocations of *THREAD_2* with the same value of *i* to the same core in a *modulo* fashion.

The macros encoding the DFPL and scheduling policy are defined outside the code of the main. These macros expand to *helper functions* invoked by the TSU at runtime to retrieve the information they encode.

5.4.5 Programming Example - LU Decomposition

In this section we present the mapping of another more complex DDM-VM application: The blocked LU decomposition. The code of the original program is shown in Figure 50. The code is composed of five nested loops that perform four basic operations on a blocked matrix. For demonstration purposes we choose the following indicative names for the computational kernels performing the operations: *diag*, *front*, *down* and *comb*.

```
float AA[ROW*COL];
float *A[TILE][TILE];
...
int i, j, k;
for (k=0; k<TILE; k++){                  // Loop1
        diag(A[k][k]);

        for (j=k+1; j<TILE; j++)          // Loop2
                front(A[k][k], A[k][j]);

        for (j=k+1; j<TILE; j++)          // Loop3
                down (A[k][k], A[j][k]);

        for (j=k+1; j<TILE; j++)          // Loop4
                for (i=k+1; i<TILE; i++) // Loop5
                  comb(A[j][k], A[k][i], A[j][i]);
}
```

Figure 50: The DDM-VM Blocked LU decomposition application - Original program code

Threads Mapping

Each of the four operations is mapped into one DDM thread and each invocation of the four threads produce one block of the matrix. The loops implementing the control-flow in the original application are mapped into five DDM threads named corresponding to the loop number.

The left side of Figure 51 shows the blocks produced by each of the operation threads (the producing thread name or first letter of the name is used to label the block) for the first iteration of LU decomposition corresponding to one iteration of the outermost loop of the original code. In every iteration, one invocation of the *diag* thread takes as input the diagonal block that corresponds to the iteration number to produce its new value. The invocations of the *front* thread produces the remaining blocks on the same row as the diagonal block. For each one of those block, it takes as input the result of the *diag* in addition to the current block to produce its new value. The *down* thread invocations operate in a similar fashion to produce the remaining blocks on the same column as the diagonal block. The *comb* thread invocations produce the rest of the blocks for that LU iteration. For every block it produces, it takes as input three blocks: the first is the current block, the second is one of the blocks produced by the *front* thread (in particular the block on the same column as the first block) and the third is one of the blocks produced by the *down* thread (in

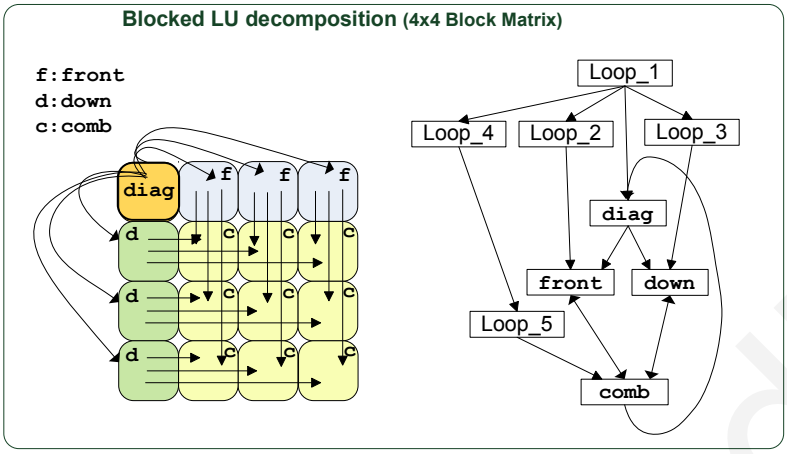


Figure 51: The DDM-VM Blocked LU decomposition application - Dependency graph

particular the block on the same row as the first block). It multiples the second and third blocks and adds the result to the first block to produce the final resulting block. The arrows in the figure indicate the input blocks needed by each thread invocation to produce its result.

Dependency Graph

This computational pattern is repeated in the next LU iteration on a subset of the resulting matrix that excludes the first row and column and continues for as much iterations as the diagonal tiles of the matrix. This results in dependencies between the thread invocations pertaining to the same LU iteration and at the same time between those and the ones pertaining to the previous iteration. The right side of Figure 51 depicts the dependency graph of all the program threads. This graph only captures a static view of the dependencies among the threads. The graph shown in Figure 52 depicts the dependencies amongst the dynamic invocations of the threads for the first two LU iterations. Each thread invocation is labelled with the value of its *context*. As mentioned previously, the invocation *context* value is derived from the values of the loop indices of the corresponding loop iteration.

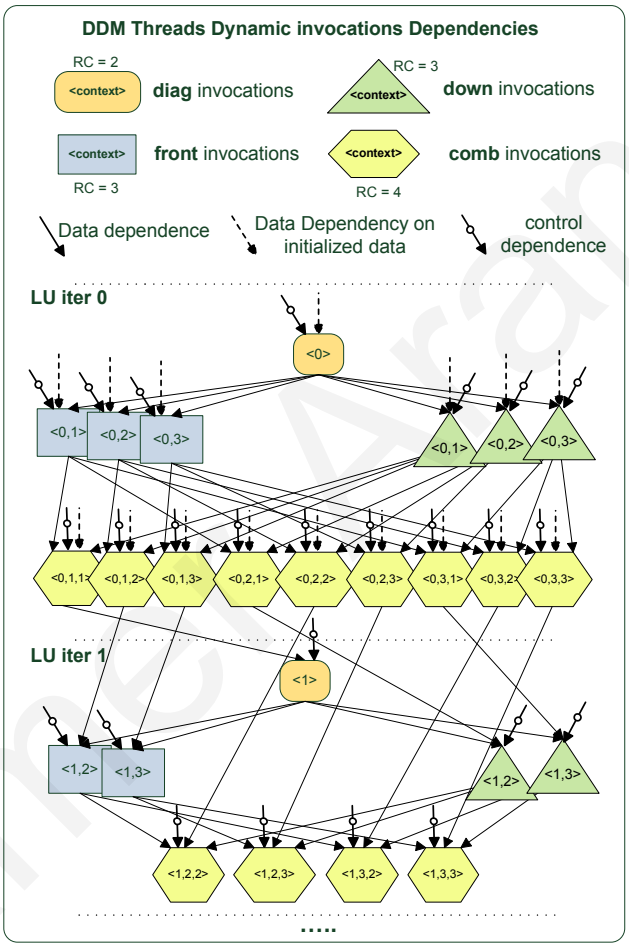


Figure 52: The DDM-VM Blocked LU decomposition application - Dependency graph among the dynamic threads invocations

Threads Code

The code of the DDM threads is shown in Figure 53. The *DVM_THREAD_START* and The *DVM_THREAD_END* marks the boundaries of the threads. The *DVM_LOOKUP* macros are used to retrieve the addresses of the input/output data and the *GET_CONTEXT* macro retrieves the components of the *context* related to the corresponding loop iteration. Each of the threads call the computational kernel responsible for performing the operation on the input/output blocks. The *DVM_UPDATE* macro is used to inform the TSU of the consumer invocation to decrement their RC. Each call of this macro corresponds to one solid dependency arrow in Figure 53. The *DVM_UPDATE* macros at the end of the *comb* thread implement a *switch actor*, which depending on the context of the invocation it updates a different consumer.

Threads Data and the *Main* Function

The macros defining the Data Frame Pointer List (DFPL) information and the *main()* of the application are shown in Figure 54-a & b, respectively. The latter includes the *DVM_SET_THREAD_TEMPLATE* macros which creates and loads the meta-data of the threads into the TSU and the *DVM_UPDATE_THREAD* macros used to decrement the RC of threads consuming initialized data. Each call of the *DVM_UPDATE_THREAD* macros corresponds to one of the dashed dependency arrows in Figure 52.

Discussion

It is imperative to note that the code of the DDM threads is directly extracted from the original program code, including the macros defining the Data Frame Pointer (DFP) information. The only part that is not directly manifested in the original code is the information regarding what consumer invocation(s) to update once a thread invocation finishes. This is implemented by the

```
DVM_THREAD_START(TID_LOOP_1);
for(k=0; k < TILE ; k++){
   DVM_UPDATE(CONS1,OP_SET_CONTEXT,MAKE_CONTEXT_S(k));
   DVM_UPDATE(CONS2,OP_SET_CONTEXT,MAKE_CONTEXT_S(k));
   DVM_UPDATE(CONS3,OP_SET_CONTEXT,MAKE_CONTEXT_S(k));
   DVM_UPDATE(CONS4,OP_SET_CONTEXT,MAKE_CONTEXT_S(k));
}
DVM_THREAD_END();
DVM_THREAD_START(TID_DIAG);
DVM_LOOKUP(float *,T);
GET_CONTEXT_S(DVM_CONTEXT,k);

diag(T); // T=A[k][k]

if (k < TILES-1)
  for (j=k+1 ; j < TILES ; j++){
   DVM_UPDATE(CONS1,OP_SET_CONTEXT_MAKE_CONTEXT_D(k,j));
   DVM_UPDATE(CONS2,OP_SET_CONTEXT_MAKE_CONTEXT_D(k,j));
  }
DVM_THREAD_END();
DVM_THREAD_START(TID_LOOP_2);
GET_CONTEXT_S(DVM_CONTEXT,k);

for(j=k+1; k < TILE ; k++)
  DVM_UPDATE(CONS1,OP_SET_CONTEXT,MAKE_CONTEXT_D(k,j));
DVM_THREAD_END();
DVM_THREAD_START(TID_FRONT);
DVM_LOOKUP(float *,T);
DVM_LOOKUP(float *,P);
GET_CONTEXT_D(DVM_CONTEXT,k,j);

front(T,P); //T=A[k][k] P=A[k][j]

for(i=k+1; i < TILES ; i++)
  DVM_UPDATE(CONS1,OP_SET_CONTEXT,MAKE_CONTEXT_T(k,i,j));
DVM_THREAD_END();
DVM_THREAD_START(TID_LOOP_3);
GET_CONTEXT_S(DVM_CONTEXT,k);

for(j=k+1; k < TILE ; k++)
  DVM_UPDATE(CONS1,OP_SET_CONTEXT,MAKE_CONTEXT_D(k,j));
DVM_THREAD_END();
DVM_THREAD_START(TID_DOWN);
DVM_LOOKUP(float *,T);
DVM_LOOKUP(float *,Q);
GET_CONTEXT_D(DVM_CONTEXT,k,j);

down(T,Q); // T=A[k][k], Q=A[j][k]
for(i=k+1; i < TILES ; i++)
  DVM_UPDATE(CONS1,OP_SET_CONTEXT,MAKE_CONTEXT_T(k,j,i));
DVM_THREAD_END();
DVM_THREAD_START(TID_LOOP_4);
GET_CONTEXT_S(DVM_CONTEXT,k);
for(j=k+1; k < TILE ; k++)
   DVM_UPDATE(CONS1,OP_SET_CONTEXT,MAKE_CONTEXT_D(k,j));
DVM_THREAD_END();
DVM_THREAD_START(TID_LOOP_5);
GET_CONTEXT_D(DVM_CONTEXT,k,j);
for(i=k+1; k < TILE ; k++)
   DVM_UPDATE(CONS1,OP_SET_CONTEXT,MAKE_CONTEXT_T(k,j,i));
DVM_THREAD_END();
DVM_THREAD_START(TID_COMB);
DVM_LOOKUP(float *,P);
DVM_LOOKUP(float *,Q);
DVM_LOOKUP(float *,S);
GET_CONTEXT_T(DVM_CONTEXT,k,j,i);

comb(P,Q,S); // P=A[j][k], Q=A[k][i], S=A[j][i]

If (j == k+1 && i == k+1) //update DIAG
 DVM_UPDATE(CONS2,OP_SET_CONTEXT,MAKE_CONTEXT_S(k+1));
else if ( j == k+1) //update FRONT
 DVM_UPDATE(CONS3,OP_SET_CONTEXT,MAKE_CONTEXT_D(j,i));
else if ( i == k+1) //update DOWN
 DVM_UPDATE(CONS4,OP_SET_CONTEXT,MAKE_CONTEXT_D(i,j));
else //update COMB
 DVM_UPDATE(CONS1,OP_SET_CONTEXT,MAKE_CONTEXT_T(k+1,j,i));
DVM_THREAD_END();
```

Figure 53: The DDM-VM Blocked LU decomposition application - The code of the DDM threads

```
DVM_CACHEFLOW_DFPL_START();
DVM_START_DFPL(THREAD_DIAG)
  GET_CONTEXT_S(DVM_CONTEXT,k);
  DVM_SET_DFP((void*)A[k][k],BS*BS*sizeof(float),DATA_READ|DATA_WRITE);
DVM_END_DFPL()

DVM_START_DFPL(THREAD_FRONT)
  GET_CONTEXT_D(DVM_CONTEXT,k,j);
  DVM_SET_DFP((void*)A[k][k],BS*BS*sizeof(float),DATA_READ);
  DVM_SET_DFP((void*)A[k][j],BS*BS*sizeof(float),DATA_READ|DATA_WRITE);
DVM_END_DFPL()

DVM_START_DFPL(THREAD_DOWN)
  GET_CONTEXT_D(DVM_CONTEXT,k,j);
  DVM_SET_DFP((void*)A[k][k],BS*BS*sizeof(float),DATA_READ);
  DVM_SET_DFP((void*)A[j][k],BS*BS*sizeof(float),DATA_READ|DATA_WRITE);
DVM_END_DFPL()

DVM_START_DFPL(THREAD_COMB)
  GET_CONTEXT_T(DVM_CONTEXT,k,j,i);
  DVM_SET_DFP((void*)A[j][k],BS*BS*sizeof(float),DATA_READ);
  DVM_SET_DFP((void*)A[k][i],BS*BS*sizeof(float),DATA_READ);
  DVM_SET_DFP((void*)A[j][i],BS*BS*sizeof(float),DATA_READ|DATA_WRITE);
DVM_END_DFPL()
DVM_CACHEFLOW_DFPL_END();
```

(a)

```
float AA[ROW*COL];
float *A[TILES][TILES]; // TILES = ROW/BSIZE;
int main(int argc, char **argv)
{
  // data initialization
  // runtime initialization

  int short ConsumerList_comb[]={THREAD_COMB,THREAD_DIAG,THREAD_FRONT,THREAD_DOWN};
  int short ConsumerList_loop_1[]={THREAD_LOOP_4,THREAD_LOOP_2,THREAD_DIAG,THREAD_LOOP_3};

  DVM_CREATE_THREAD_TEMPLATE(THREAD_DIAG   ,2,2,THREAD_FRONT,THREAD_DOWN,1,DVM_RROBIN,0,DVM_ARITY_1,
                          SM_PERFECT, MAKE_CONTEXT_S(make_mask(TILE-1)));

  DVM_CREATE_THREAD_TEMPLATE(THREAD_FRONT  ,3,1,THREAD_COMB,0           ,2,DVM_RROBIN,0,DVM_ARITY_2,
                          SM_PERFECT,MAKE_CONTEXT_D(make_mask(TILE-1),make_mask(TILE-1)));

  DVM_CREATE_THREAD_TEMPLATE(THREAD_DOWN   ,3,1,THREAD_COMB,0           ,2,DVM_RROBIN,0,DVM_ARITY_2,
                          SM_PERFECT,MAKE_CONTEXT_D(make_mask(TILE-1),make_mask(TILE-1)));

  DVM_CREATE_THREAD_TEMPLATE(THREAD_COMB   ,4,4,0,ConsumerList_Comb     ,3,DVM_RROBIN,0,DVM_ARITY_3,
                          SM_PERFECT,MAKE_CONTEXT_T(make_mask(TILE-1),make_mask(TILE-1),make_mask(TILE-1)));

  DVM_CREATE_THREAD_TEMPLATE(THREAD_LOOP_1 ,1,4,0,ConsumerList_loop_1   ,0,DVM_RROBIN,0,DVM_ARITY_0,
                          SM_PERFECT,MAKE_CONTEXT_S(make_mask(1)));

  DVM_CREATE_THREAD_TEMPLATE(THREAD_LOOP_2 ,1,1,THREAD_FRONT ,0,0,DVM_RROBIN,0,DVM_ARITY_1,
                                  SM_PERFECT,MAKE_CONTEXT_S(make_mask(TILE-1)));
  DVM_CREATE_THREAD_TEMPLATE(THREAD_LOOP_3 ,1,1,THREAD_DOWN  ,0,0,DVM_RROBIN,0,DVM_ARITY_1,
                                  SM_PERFECT,MAKE_CONTEXT_S(make_mask(TILE-1)));
  DVM_CREATE_THREAD_TEMPLATE(THREAD_LOOP_4 ,1,1,THREAD_LOOP_5,0,0,DVM_RROBIN,0,DVM_ARITY_1,
                                  SM_PERFECT,MAKE_CONTEXT_S(make_mask(TILE-1)));
  DVM_CREATE_THREAD_TEMPLATE(THREAD_LOOP_5 ,1,1,THREAD_COMB,0,0,DVM_RROBIN,0,DVM_ARITY_2,
                                  SM_PERFECT,MAKE_CONTEXT_D(make_mask(TILE-1),make_mask(TILE-1)));

  DVM_UPDATE_THREAD(THREAD_DIAG,0);

  for (j=1 ; j < TILE ;j++)
  {
        DVM_UPDATE_THREAD(THREAD_FRONT,MAKE_CONTEXT_D(0,j));
        DVM_UPDATE_THREAD(THREAD_DOWN ,MAKE_CONTEXT_D(0,j));
       for(i=1 ; i < TILES ; i++)
              DVM_UPDATE_THREAD(THREAD_COMB ,MAKE_CONTEXT_T(0,j,i);
  }

  DVM_UPDATE_THREAD(THREAD_LOOP_1,0);

  DVM_EXECUTE(); // this will block until all threads finish execution

  // verification, use results here

}
```

(b)

Figure 54: The DDM-VM Blocked LU decomposition application. (a) The DFPL definition macros (b) The main() function

the *DVM_UPDATE* macros and the consumer identifier and *context* information they convey. This information is extracted by analyzing the data consumed/produced by the different threads, a task that can be performed manually by the programmer or automatically by the compiler using techniques similar to the ones utilized in [98]. In the Section 5.8 we present the ongoing efforts on the automatic generation of DDM code using the GCC compiler. In Section 6.1, we present a technique that can be used when it is not possible to extract the dependency information at compile-time.

Programs with this level of complexity are hard to express using current parallel programming models and languages. The programmers would either resort to complex parallel synchronization constructs like locks, semaphores and barriers or use a more automated approach like OpenMP, which cannot extract all the parallelism in such applications. On the other hand, the data-driven approach adopted by DDM-VM can naturally express the parallelism in the program. Even if this approach requires the programmer to reason about the program and analyze the dependencies (in the absence of a compilation tools), the programmer is spending his or her time thinking in terms of the algorithm itself as opposed to the low-level parallelization details, the synchronization constructs or race conditions, etc. Moreover, in the case of architectures with software-managed memory hierarchy, the programmer is relieved from tackling the issues of data allocation & movement in the memory hierarchy and structuring the code to benefit from double buffering, etc. All this underscore the benefits of the programming approach we are proposing in this work.

5.5 Supporting Distributed DDM Execution

In this section we discuss the support for distributed DDM execution. We re-visit the elements of the programming methodology and discuss the changes needed to support distributed execution.

5.5.1 DDM-VM Macros

Supporting distributed execution introduces two minor changes to the DDM-VM macros. The first is more of a restriction than a change, which is related to the scheduling policies that can be assigned to threads. As explained in Section 4.1 the work in this thesis explores a *static* thread mapping scheme. Consequently, the programmer can use the *static* and *modulo* scheduling policies in a distributed DDM-VM program. In addition, the *custom* policy can be used as long as the defined mapping does not change at runtime. The rest of the policies (i.e. *dynamic* and *roundrobin*) are not supported.

The second change is related to the macro that encodes the thread DFP information. The *address* field in the macro is changed to refer to a GAS address instead of a conventional memory address, i.e., it is composed of the ordered pair **(node_id,local_address)**. Moreover, a new field called *forward-list* is added to the macro to support forwarding the produced data to multiple nodes in the case of threads that have multiple remote consumers. The GAS addresses in the *forward-list* are stored at runtime in the Forward Table (FT) inside the TSU.

5.5.2 Data Distribution

Distributed execution entails the distribution of the program data across the nodes. In particular, the data of a certain invocations must be mapped to the part of the GAS located on the node where this invocations is scheduled to run.

The DDM-VM provides a set of runtime library calls that help with the initial allocation/distribution of data. Furthermore, the runtime facilitates the gathering of data at a specific node after the program finishes execution, which is typically desired for processing the results or exporting them to disk. These calls internally invoke the services of the NIU to move the data between the main memories of the nodes. The calls are partly inspired by the ones provided by P-GAS languages like Unified Parallel C (UPC)[25] as they tackle the same issues of allocation and data distribution within a global address space. However, we only provide a subset of the abstractions these languages provide, as they typically utilize specialized compilers that perform many automatic operations on the data to support such abstractions. However, the subset we provide is sufficient to perform all the tasks at hand. The calls are categorized into three types: data allocation, data movement and utility routines. Please refer to Appendix A for a detailed description of these runtime calls.

Overall, programming for distributed execution is fundamentally the same as within a node, however, more attention is given to the distribution of data and threads across the cores since the cost of communication is much larger than that in the case of single-node execution.

Next, we re-visit the LU decomposition DDM-VM program and adapt it to support distributed DDM execution.

5.5.3 LU Decomposition - Distributed Version

The principal concern when adapting programs for distributed execution is the distribution of thread invocations and data across the nodes. In the case of the LU program, the majority of the program threads are assigned the *modulo* policy to spread the invocations across all the cores in the system. Therefore, almost all of the matrix blocks are required on all the nodes. Hence, we apply a commonly utilized technique in distributed programming, in which we allocate a copy of

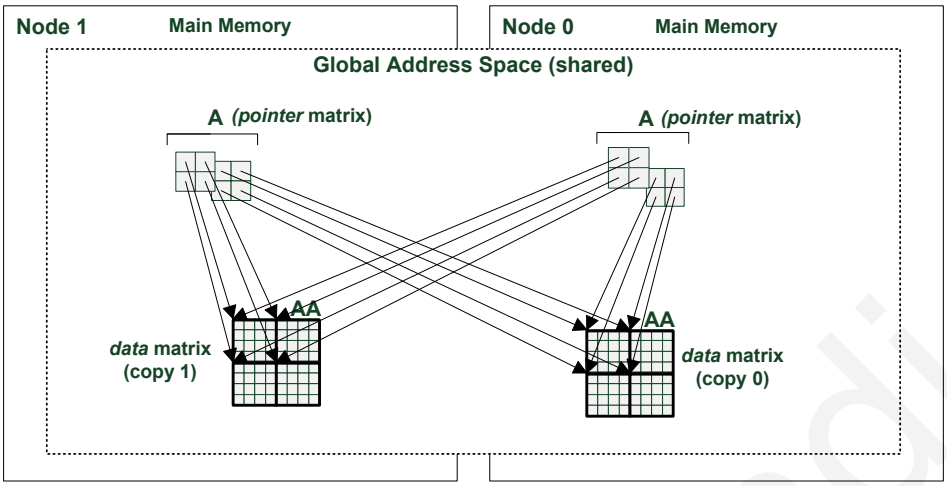


Figure 55: Memory Layout for the LU Program - a System with Two Nodes and a 4x4 Blocked Matrix

the matrix on each node. This simplifies the initial distribution step into the mere copying of the matrix from the root node (after loading the matrix from disk) to the rest of the nodes. Note that since the matrix copies are allocated in the GAS, they are visible to all the nodes and every node allocates an auxiliary pointer matrix to reference all the copies. Figure 55 shows an example of the data layout for the LU program on a system with two nodes and a 4x4 blocked matrix. For every node, *AA* is the blocked matrix holding the data and *A* is the auxiliary pointer matrix holding pointers to both the local copy of *AA* and the remote one on the other node.

Figure 56 depicts the code of the main() function. Comparing the code of the main in this figure to the one illustrated previously in Figure 54-b, three changes in the code can be noticed, which we highlight using shaded boxes. The first is the addition of a data distribution step that takes place after the initialization. In this step the data matrix is distributed and the addresses are communicated across the nodes. The second change is the setting of the scheduling policy of the threads to *static* and *modular*, since the previously assigned *roundrobin* policy is not supported in distributed execution. The third change is the addition of a *gather_data()* function, which purpose will be explained further on.

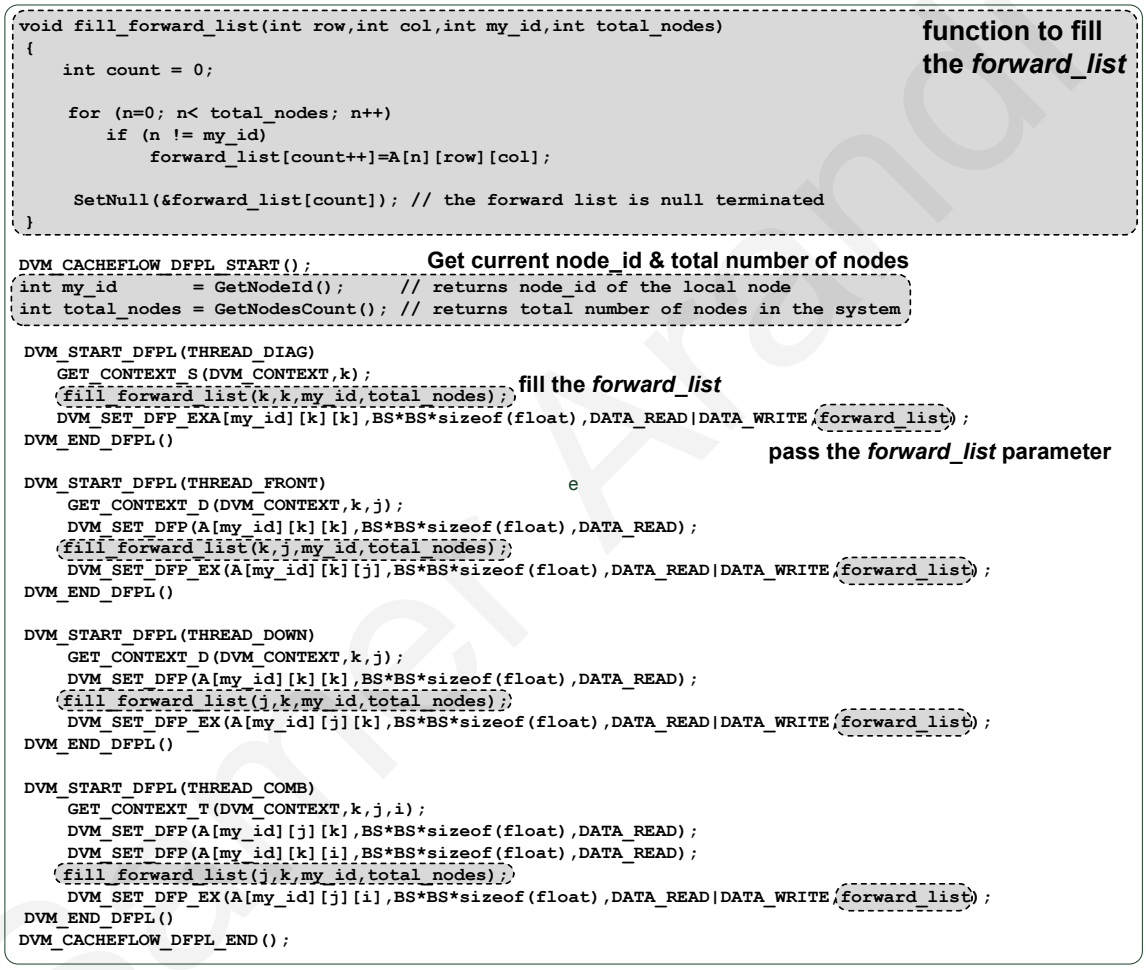
```
float AA[ROW*COL]; // local array holding the data, TILES = ROW/BSIZE; BSIZE: the block dimension
g_address ***A;    // auxiliary pointer matrices
int main(int argc, char **argv)
{
  // data initialization
  // runtime initialization                                              distribution of data
 my_id = GetNodeId();

 // exchange the pointers among the nodes
 for (n = 0 ; n < nodes_num ; n++)
       for (m = 0 ; m < nodes_num ; m++)
          if ( m != n )  // I don't want to send to myself!
               for (i = 0 ; i < TILES ; i++)
                  dvm_move(A[n][i],A[m][i],TILES*sizeof(g_address));

 // copy the data array from the root to the rest of the nodes
 for (n = 1 ; n < nodes_num ; n++)
       dvm_move(A[0][0][0],A[n][0][0],ROW*COL*sizeof(float));

  int short ConsumerList_comb[]={THREAD_COMB,THREAD_DIAG,THREAD_FRONT,THREAD_DOWN};
  int short ConsumerList_loop_1[]={THREAD_LOOP_4,THREAD_LOOP_2,THREAD_DIAG,THREAD_LOOP_3};

  DVM_CREATE_THREAD_TEMPLATE(THREAD_DIAG  ,…,DVM_STATIC,0,…);          Using static and
  DVM_CREATE_THREAD_TEMPLATE(THREAD_FRONT ,…,DVM_MODULAR,MASK_INDX,…); modulo policies
  DVM_CREATE_THREAD_TEMPLATE(THREAD_DOWN  ,…,DVM_MODULAR,MASK_INDX,…);
  DVM_CREATE_THREAD_TEMPLATE(THREAD_COMB  ,…,DVM_MODULAR,MASK_CNTX,…);

  DVM_CREATE_THREAD_TEMPLATE(THREAD_LOOP_1,…,DVM_STATIC,0,…);
  DVM_CREATE_THREAD_TEMPLATE(THREAD_LOOP_2,…,DVM_MODULAR,MASK_INDX,…);
  DVM_CREATE_THREAD_TEMPLATE(THREAD_LOOP_3,…,DVM_MODULAR,MASK_INDX,…);
  DVM_CREATE_THREAD_TEMPLATE(THREAD_LOOP_4,…,DVM_MODULAR,MASK_INDX,…);
  DVM_CREATE_THREAD_TEMPLATE(THREAD_LOOP_5,…,DVM_MODULAR,MASK_INDX,…);

  DVM_UPDATE_THREAD(THREAD_DIAG,0);

  for (j=1 ; j < TILE ;j++)
  {
       DVM_UPDATE_THREAD(THREAD_FRONT,MAKE_CONTEXT_D(0,j));
       DVM_UPDATE_THREAD(THREAD_DOWN ,MAKE_CONTEXT_D(0,j));
       for(i=1 ; i < TILES ; i++)
              DVM_UPDATE_THREAD(THREAD_COMB ,MAKE_CONTEXT_T(0,j,i);
  }

  DVM_UPDATE_THREAD(THREAD_LOOP_1,0);

  DVM_EXECUTE(); // this will block until all threads finish execution   gathering of data
  gather_data(0); // collect gather the results from all the nodes to the root node

  // verification, use results here
}
```

Figure 56: Distributed DDM-VM LU Program - main() function

Data Forwarding

The forwarding of data between producers and remote consumers during the execution is performed automatically by the DDM-VM, however, the programmer is responsible for assigning the addresses of the produced data on the remote node for the DDM-VM to be able to perform this task. The programmer conveys this information in the DFPL macros specifying the input/output of the threads.

To facilitate this task we adopt the following simple convention: every producer that potentially has multiple remote consumers forwards the produced data to all the nodes via the *forward_list* parameter of the DFP macro. The DDM-VM is cleverly designed so that only the *forwards* that correspond to actual remote consumers are executed. This is applied when the RC request is to be sent to a remote consumer. The TSU checks the Forward Table (FT) and only the forwards with a destination address belonging to the remote node are piggy-backed in the same message. When the thread finishes execution the TSU discards any *redundant* forwards. Using this simple technique the forwarding mechanism is tremendously simplified without suffering any overheads. Figure 57 illustrates the DFPL definition macros for the LU application. The figure is better understood when compared with Figure 54-a of the single-node LU application. The only differences (indicated via shaded boxes) are the addition of the *forward_list* parameter to the extended variant of the *DVM_SET_DFP*, which we call *DVM_SET_DFP_EX* and the use of GAS addresses instead of conventional memory addresses to refer to data. The *forward_list* parameter is filled using the *fill_forward_list* function.




```
void fill_forward_list(int row,int col,int my_id,int total_nodes)
 {
    int count = 0;

    for (n=0; n< total_nodes; n++)
       if (n != my_id)
           forward_list[count++]=A[n][row][col];

    SetNull(&forward_list[count]); // the forward list is null terminated
 }

DVM_CACHEFLOW_DFPL_START();
int my_id       = GetNodeId();     // returns node_id of the local node
int total_nodes = GetNodesCount(); // returns total number of nodes in the system

 DVM_START_DFPL(THREAD_DIAG)
    GET_CONTEXT_S(DVM_CONTEXT,k);
    fill_forward_list(k,k,my_id,total_nodes);
    DVM_SET_DFP_EXA[my_id][k][k],BS*BS*sizeof(float),DATA_READ|DATA_WRITE,forward_list);
 DVM_END_DFPL()

 DVM_START_DFPL(THREAD_FRONT)
     GET_CONTEXT_D(DVM_CONTEXT,k,j);
     DVM_SET_DFP(A[my_id][k][k],BS*BS*sizeof(float),DATA_READ);
    fill_forward_list(k,j,my_id,total_nodes);
     DVM_SET_DFP_EX(A[my_id][k][j],BS*BS*sizeof(float),DATA_READ|DATA_WRITE,forward_list);
 DVM_END_DFPL()

 DVM_START_DFPL(THREAD_DOWN)
     GET_CONTEXT_D(DVM_CONTEXT,k,j);
     DVM_SET_DFP(A[my_id][k][k],BS*BS*sizeof(float),DATA_READ);
    fill_forward_list(j,k,my_id,total_nodes);
     DVM_SET_DFP_EX(A[my_id][j][k],BS*BS*sizeof(float),DATA_READ|DATA_WRITE,forward_list);
 DVM_END_DFPL()

 DVM_START_DFPL(THREAD_COMB)
     GET_CONTEXT_T(DVM_CONTEXT,k,j,i);
     DVM_SET_DFP(A[my_id][j][k],BS*BS*sizeof(float),DATA_READ);
     DVM_SET_DFP(A[my_id][k][i],BS*BS*sizeof(float),DATA_READ);
    fill_forward_list(j,k,my_id,total_nodes);
     DVM_SET_DFP_EX(A[my_id][j][i],BS*BS*sizeof(float),DATA_READ|DATA_WRITE,forward_list);
 DVM_END_DFPL()
DVM_CACHEFLOW_DFPL_END();
```

Figure 57: Distributed DDM-VM LU Program - DFPL Definition

```
void gather_data(int root_id)
{
   int i,j,k,core_id,node_id,my_id,cores_count;
   my_id       = GetNodeId();
   cores_count = GetCoresCount();

   for (k = 0 ; k < DIM ; k++)
     for (i = k+1 ; i < DIM ; i++)
     {
       // this is for the front thread
       core_id = i % cores_count; // get the core to which the invocation of front that
        // produced this tile is scheduled
       node_id = GetCoreNodeId(core_id); // get the node which this core belongs to

       // get this tile if it was produced by other node than the root and its not a diagonal tile
       if ( node_id != root_id && i != k)
            dvm_move(A[node_id][k][i],A[root_id][k][i],BSIZE*BSIZE*sizeof(DATA_TYPE));

       // this is for the down thread
       core_id = i % cores_count; // get the core to which the invocation of down that
        // produced this tile is scheduled
       node_id = GetCoreNodeId(core_id); // get the node which this core belongs to

       // get this tile if it was produced by other node than the root and its not a diagonal tile
       if ( node_id != root_id && i != k)
            dvm_move(A[node_id][i][k],A[root_id][i][k],BSIZE*BSIZE*sizeof(DATA_TYPE));

     }
}
```

Figure 58: Distributed DDM-VM LU Program - gather_data() function

Gathering Data

Gathering the result data into the root is performed by the *gather_data()* function called right after the *DDM_EXECUTE* macro in the main() function shown at Figure 56. The code of this function is depicted in Figure 58. This function iterates over the tiles of the matrix and copies the tile from the node where it was lastly updated to the corresponding tile position on the matrix located on the root node.

The function makes use of the modulo % operator to find out the core where each tile was produced and then using the **`GetCoreNodesId()`** utility function determines if the core belongs to a node other than the root and if so the tile is copied.

The rest of the DDM-VM program including the code of the threads and the dependency graph is not changed. Moreover, this program supports any number of nodes and any number of cores

per node without the need for any change. In fact, this program runs normally on a single-node as well.

5.6 DDM-VM Optimizations

In this section we discuss some of the optimizations deployed to improve the DDM-VM performance and manage the system resources.

5.6.1 Consumer Updating Optimizations

Incremental Update Optimization

If we examine the code of *THREAD_2* in Figure 45 we notice that each invocation of this thread (except for the last one) updates the consuming invocation that corresponds to the iteration with the same outer-loop index and the next inner loop index. In other words the consumer invocation has the same *context* value as the producer invocation except for the lower part, which is at a distance of +1. This pattern occurs frequently in threads implementing loops where the consumer invocation corresponds to the next inner or outer loops iteration. In such cases instead of creating the whole *context* value every time and communicating it to the TSU we use a set of operators that inform the TSU the relation between the producer invocation and the consumer invocation so that the TSU deduces the value of the latter. This optimization can be utilized when the size of the *context* is large and the communication mechanism between the cores and the TSU is expensive. The *DVM_UPDATE(CONS,OP,VALUE)* macro implements this optimization via the *OP* and *VALUE* parameters. The possible values of *OP* parameters include:

- OP_SET_CONTEXT: informs the TSU to decrement the RC of consumer *CONS* with a *context* equal to *VALUE*, i.e., the optimization is not applied here.

- OP_INC_INDX: informs the TSU to decrement the RC of of consumer *CONS* with a *context* that is equal to the current producer thread *context* except that the part corresponding the to the most-inner loop is incremented by VALUE+1. This is used with threads that has an *arity*

of 1, 2 or 3, i.e., threads implementing one-level loop, two-level or three-level nested loops. The reason we increment by Value+1, is so that in the default case (when VALUE=zero) the increment amount is 1, which is the more general case.

- OP_INC_CNTX: informs the TSU to decrement the RC of of consumer *CONS* with a *context* that is equal to the current producer thread *context* except that the part corresponding the to the outer-most loop is incremented by VALUE+1. This is used with threads that has an *arity* of 2 or 3.

- OP_INC_INDX2: informs the TSU to decrement the RC of of consumer *CONS* with a *context* that is equal to the current producer thread *context* except that the part corresponding the to the 2nd most-inner loop is incremented by VALUE+1. This is used with threads that has an *arity* of 3.

- OP_NOP: informs the TSU to decrement the RC of of consumer *CONS* with a *context* that is equal to the current producer thread *context*.

- OP_FIN: informs the TSU that this update request is the last for the currently executing thread and so the thread has finished execution. This flag is an optimization that piggy-backs this notification on the update request. If this flag is not set by the programmer, the *DVM_THREAD_END()* macro located at end of the thread code automatically informs the TSU that the thread finished.

Note that if *CONS* is set to *CONS_NONE* no update operation occurs. Moreover, *VALUE* can be negative to cater for the cases when a loop proceeds in the reverse order. Figure 59 shows the result of applying this optimization to the code of *THREAD_2*.

```
DVM_THREAD_START(TID_THREAD_2);
DVM_LOOKUP(float *,A);
DVM_LOOKUP(float *,B);
DVM_LOOKUP(float *,C);
GET_CONTEXT_D(DVM_CONTEXT,i,j);

MultAdd_Block(C,A,B);
    DVM_UPDATE(CONS1,OP_INC_INDX,1);
if (j < N-1)

DVM_THREAD_END();
```

Figure 59: The code of *THREAD_2* of the blocked matrix multiplication DDM-VM program, shown previously in Figure 45, after applying the incremental update optimization

Compound Update Optimization

Another optimization with a more substantial effect on the performance is the *compound update* optimization. To understand this optimization, we examine the code of *THREAD_1* in Figure 45 and the code of the *diag*, *front* and *down* threads in Figure 53. We find that for each iteration of the enclosing loop a call is made to the *DVM_UPDATE* macro to update the RC of one invocation of the consumer(s) threads. For every such call a message is sent to the TSU and an entry is inserted in one of the TSU Queues. This pattern where consecutive invocations of a consumer thread are updated occurs frequently in DDM-VM programs. To optimize this operation the special macro *DVM_UPDATE_MULTIPLE* is provided to send a special request to the TSU for decrementing multiple consecutive invocations of a consumer thread. The TSU manages this special request internally in an optimized manner, which reduces overheads significantly. Using this macro doesn't change the semantics of the program nor the synchronization graph. Figure 60 shows the result of applying this optimization to the code of *THREAD_1*.

```
DVM_THREAD_START(TID_THREAD_1);
  DVM_UPDATE_MULTIPLE(CONS1,MAKE_CONTEXT_D(0,0),
                            MAKE_CONTEXT_D(M-1,0);

DVM_THREAD_END();
```

Figure 60: The code of *THREAD_1* of the blocked matrix multiplication DDM-VM program, shown previously in Figure 45, after applying the compound update optimization

Figure 61 shows the result of applying this optimization to the code of the *diag*, *front* and *down* threads.

```
DVM_THREAD_START(TID_DIAG);
DVM_LOOKUP(float *,T);
GET_CONTEXT_S(DVM_CONTEXT,k);

diag(T); // T=A[k][k]

if (k < TILES-1){
   DVM_UPDATE_MULTIPLE(CONS1,MAKE_CONTEXT_D(k,k+1),
                             MAKE_CONTEXT_D(k,TILES-1));
   DVM_UPDATE_MULTIPLE(CONS2,MAKE_CONTEXT_D(k,k+1),
                             MAKE_CONTEXT_D(k,TILES-1));
}
DVM_THREAD_END();
DVM_THREAD_START(TID_FRONT);
DVM_LOOKUP(float *,T);
DVM_LOOKUP(float *,P);
GET_CONTEXT_D(DVM_CONTEXT,k,j);

front(T,P); //T=A[k][k] P=A[k][j]

DVM_UPDATE_MULTIPLE(CONS1,MAKE_CONTEXT_T(k,k+1,j),
                          MAKE_CONTEXT_T(k,TILES-1 ,j));
DVM_THREAD_END();
DVM_THREAD_START(TID_DOWN);
DVM_LOOKUP(float *,T);
DVM_LOOKUP(float *,Q);
GET_CONTEXT_D(DVM_CONTEXT,k,j);

down(T,Q); // T=A[k][k], Q=A[j][k]

DVM_UPDATE_MULTIPLE(CONS1,MAKE_CONTEXT_T(k,j,k+1),
                          MAKE_CONTEXT_T(k,j,TILES-1));
DVM_THREAD_END();
```

Figure 61: The code of *diag*, *front* and *down* threads of the blocked LU decomposition DDM-VM program, shown previously in Figure 53, after applying the compound update optimization

The *DVM_UPDATE_MULTIPLE(CONS,VALUE1,VALUE2)* takes three parameters, the first one specifies the consumer thread and the 2nd and 3rd ones specify, respectively, the start and end bounds of the consecutive range of consumer thread invocations to decrement their RC. The TSU extracts the bounds from the two values according to the *arity* of the consumer thread. For example the *DVM_UPDATE_MULTIPLE* macro in the optimized *THREAD_1* code is used to decrement the RC of the invocations of *THREAD_2*, which has an *arity* of 2, since it implements a two-level nested loop. Therefore, the *MAKE_CONTEXT_D* macro is used to create the bound values of the targeted invocations. The bounds specify invocations with the following consecutive *context* values: <0,0>,<1,0>,<2,0>,..., <M-1,0>. On the other hand, *DVM_UPDATE_MULTIPLE* macro

in the optimized code of the *front* thread is used to decrement the RC of the invocations of the *comb* thread, which has an *arity* of 3. Therefore, the *MAKE_CONTEXT_T* macro is used to specify the invocations with the following consecutive *context* values:

<k,k+1,j>,<k,k+2,j>,<k,k+3,j>,...,<k,TILES-1,j>.

5.6.2 Eliminating Redundant Dependencies

As noted in Section 5.4.1 when mapping code containing loops to DDM threads, in many programs, the loop indices directly correspond to the mapped threads *context* value. As the loop indices are extracted from the *context* in this case, it is possible to remove the loop index maintenance code. In effect this results in removing the now redundant control-flow dependencies originally driving the program execution, since the data-dependencies are enough to drive the execution in the mapped program.

This optimization considerably cuts down the amount of dependencies in the program and effectively the operations performed during the execution, thus possibly reducing the execution time, power consumption and other system resources.

Upon revisiting The LU decomposition in Section 5.4.5, it is clear from studying the dependency graph in Figure 52 that even if we remove the control-flow dependencies there is at least one incoming data-dependence arrow at each thread invocation, i.e., the data-dependencies are enough to drive the execution. To apply the proposed optimization, the threads implementing the loops iterators (threads with the prefix LOOP) are removed and the the RC of the rest of the threads (threads consuming and producing data) is reduced by 1 to account for the removed control-flow dependency.

Figure 62 depicts the full code of the LU decomposition program after applying this optimization in addition to the optimization of Section 5.6.1.

```
float AA[ROW*COL];
float *A[TILES][TILES]; // TILES = ROW/BSIZE;
int main(int argc, char **argv)
{

// data initialization
// runtime initialization

int short ConsumerList[]={THREAD_COMB,THREAD_DIAG,
                          THREAD_FRONT,THREAD_DOWN};

DVM_CREATE_THREAD_TEMPLATE(THREAD_DIAG, 2,2,
                        THREAD_FRONT,THREAD_DOWN,1,
                        DVM_RROBIN,0,DVM_ARITY_1,
                        SM_PERFECT,
                        MAKE_CONTEXT_S(make_mask(DIM-1)));

DVM_CREATE_THREAD_TEMPLATE(THREAD_FRONT,3,1,
                        THREAD_COMB,0,2,
                        DVM_RROBIN,0,DVM_ARITY_2,
                        SM_PERFECT,
MAKE_CONTEXT_D(make_mask(DIM-1),make_mask(DIM-1)));

DVM_CREATE_THREAD_TEMPLATE(THREAD_DOWN, 3,1,
                        THREAD_COMB,0,2,
                        DVM_RROBIN,0,DVM_ARITY_2,
                        SM_PERFECT,
MAKE_CONTEXT_D(make_mask(DIM-1),make_mask(DIM-1)));

DVM_CREATE_THREAD_TEMPLATE(THREAD_COMB, 4,4,
                        0,ConsumerList_Comb,3,
                        DVM_RROBIN,0,DVM_ARITY_3,
                          SM_PERFECT,
MAKE_CONTEXT_T(make_mask(DIM-1),
               make_mask(DIM-1),make_mask(DIM-1)));


DVM_UPDATE_THREAD(THREAD_DIAG,0);
DVM_UPDATE_THREAD_MULTIPLE(THREAD_FRONT,
     MAKE_CONTEXT_D(0,1),MAKE_CONTEXT_D(0,TILES-1));

DVM_UPDATE_THREAD_MULTIPLE(THREAD_DOWN,
     MAKE_CONTEXT_D(0,1),MAKE_CONTEXT_D(0,TILES-1));

DVM_UPDATE_THREAD_MULTIPLE(THREAD_COMB,
MAKE_CONTEXT_T(0,1,1),MAKE_CONTEXT_T(0,TILES-1,TILES-1));


DVM_EXECUTE();//block until all threads finish

// verification, use results here
}
```

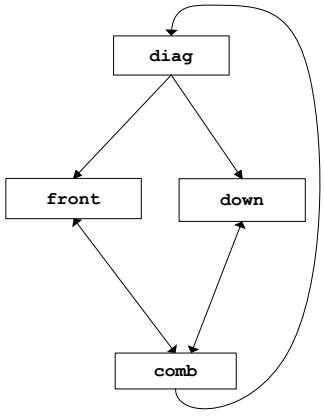
(a)

```
DVM_THREAD_START(TID_DIAG);
DVM_LOOKUP(float *,T);
GET_CONTEXT_S(DVM_CONTEXT,k);

diag(T); // T=A[k][k]

if (k < TILES-1){
 DVM_UPDATE_MULTIPLE(CONS1,MAKE_CONTEXT_D(k,k+1),
                     MAKE_CONTEXT_D(k,TILES-1));
 DVM_UPDATE_MULTIPLE(CONS2,MAKE_CONTEXT_D(k,k+1),
                     MAKE_CONTEXT_D(k,TILES-1));
}
DVM_THREAD_END();
DVM_THREAD_START(TID_FRONT);
DVM_LOOKUP(float *,T);
DVM_LOOKUP(float *,P);
GET_CONTEXT_D(DVM_CONTEXT,k,j);

front(T,P); //T=A[k][k] P=A[k][j]

DVM_UPDATE_MULTIPLE(CONS1,MAKE_CONTEXT_T(k,k+1,j),
                    MAKE_CONTEXT_T(k,TILES-1 ,j));
DVM_THREAD_END();
DVM_THREAD_START(TID_DOWN);
DVM_LOOKUP(float *,T);
DVM_LOOKUP(float *,Q);
GET_CONTEXT_D(DVM_CONTEXT,k,j);

down(T,Q); // T=A[k][k], Q=A[j][k]

DVM_UPDATE_MULTIPLE(CONS1,MAKE_CONTEXT_T(k,j,k+1),
                    MAKE_CONTEXT_T(k,j,TILES-1));
DVM_THREAD_END();
DVM_THREAD_START(TID_COMB);
DVM_LOOKUP(float *,P);
DVM_LOOKUP(float *,Q);
DVM_LOOKUP(float *,S);
GET_CONTEXT_T(DVM_CONTEXT,k,j,i);

comb(P,Q,S); // P=A[j][k], Q=A[k][i], S=A[j][i]

// switch actor
If (j == k+1 && i == k+1) //update DIAG
 DVM_UPDATE(CONS2,OP_SET_CONTEXT,MAKE_CONTEXT_S(k+1));
else if ( j == k+1) //update FRONT
 DVM_UPDATE(CONS3,OP_SET_CONTEXT,MAKE_CONTEXT_D(j,i));
else if ( i == k+1) //update DOWN
 DVM_UPDATE(CONS4,OP_SET_CONTEXT,MAKE_CONTEXT_D(i,j));
else //update COMB
 DVM_UPDATE(CONS1,OP_SET_CONTEXT,MAKE_CONTEXT_T(k+1,j,i));
DVM_THREAD_END();
```

(b)

```
DVM_CACHEFLOW_DFPL_START();
 DVM_START_DFPL(THREAD_DIAG)
   GET_CONTEXT_S(DVM_CONTEXT,k);
   DVM_SET_DFP((void*)A[k][k],BS*BS*sizeof(float),DATA_READ|DATA_WRITE);
 DVM_END_DFPL()

 DVM_START_DFPL(THREAD_FRONT)
   GET_CONTEXT_D(DVM_CONTEXT,k,j);
   DVM_SET_DFP((void*)A[k][k],BS*BS*sizeof(float),DATA_READ);
   DVM_SET_DFP((void*)A[k][j],BS*BS*sizeof(float),DATA_READ|DATA_WRITE);
 DVM_END_DFPL()

 DVM_START_DFPL(THREAD_DOWN)
   GET_CONTEXT_D(DVM_CONTEXT,k,j);
   DVM_SET_DFP((void*)A[k][k],BS*BS*sizeof(float),DATA_READ);
   DVM_SET_DFP((void*)A[j][k],BS*BS*sizeof(float),DATA_READ|DATA_WRITE);
 DVM_END_DFPL()

 DVM_START_DFPL(THREAD_COMB)
   GET_CONTEXT_T(DVM_CONTEXT,k,j,i);
   DVM_SET_DFP((void*)A[j][k],BS*BS*sizeof(float),DATA_READ);
   DVM_SET_DFP((void*)A[k][i],BS*BS*sizeof(float),DATA_READ);
   DVM_SET_DFP((void*)A[j][i],BS*BS*sizeof(float),DATA_READ|DATA_WRITE);
 DVM_END_DFPL()
DVM_CACHEFLOW_DFPL_END();
```

(b)



(d)

Figure 62: The DDM-VM Blocked LU decomposition application after optimization. (a) The main() function code (b) The code of the DDM threads (c) The DFPL definition macros (d) the dependency graph

It is important to note that in certain cases we do not employ this optimization even if it is applicable. In the case of programs with data-dependencies that cannot be uncovered at compile time, the technique we utilize for handling such programs (described in Chapter 6) requires the existence of at least one explicit dependency. In many cases the only explicit dependency is a control-flow dependency and so it cannot be removed. Moreover, we have found that, in general, applying the loop *throttling* optimization (described in the next section) is more convenient and straightforward to implement in the presence of the control-flow dependencies.

5.6.3 Resource Management

Unlike other techniques that have difficulty extracting parallelism, Dynamic Data-Flow based techniques have the property of exposing the maximum potential parallelism which could overwhelm the resources of the machine. This is a classical problem in Data-Flow execution [27, 103]. In DDM-VM the most critical resources are the TSU and the caches. DDM-VM controls the amount of concurrency to match the available resources both implicitly by the TSU and explicitly at the level of DDM-VM programs.

TSU Resource Management

The operations of the TSU matches the status of the TSU resources: TSU queues and structures. When the Extended Firing Queue (ExFQ) holding the information of ready threads is full or the Local Store (LS) memory of a certain SPE is full, the S-CacheFlow module in the TSU is disabled until some resources are freed. This type of control mechanism is transparent by default, but can be controlled using the DDM-VM configuration files.

Resource Management in Programs

The programmer can apply a technique similar to loop throttling (bounding) [27] to limit the number of thread invocations that are active concurrently. This technique is implemented by introducing auxiliary dependencies in the program graph.

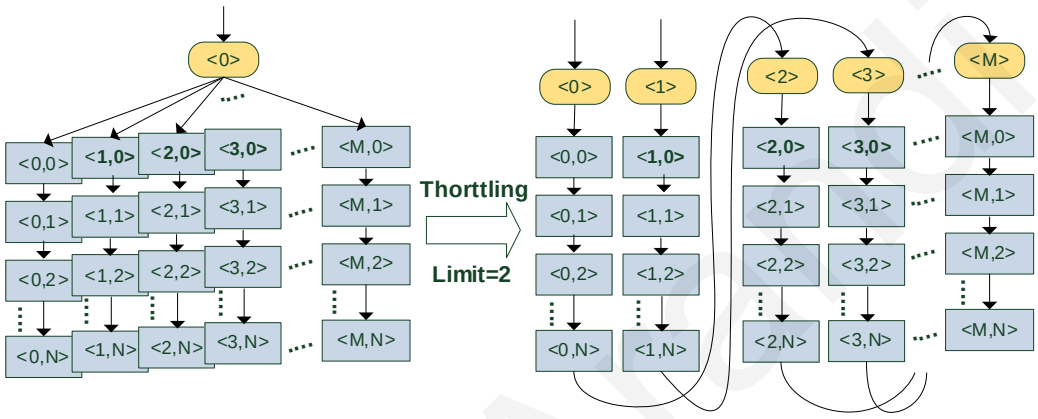


Figure 63: Resource Management - Throttling with limit set to 2

Figure 63 illustrates the dynamic dependency graph for the example in Figure 44 after applying this technique on *THREAD_1* implementing the outer loop. The limit value for the throttling is set to 2 in this example. This value can be determined at run-time and adapted to the problem size and number of execution units.

Another technique is to partition the program into DDM Blocks. A DDM block is equivalent to a function or a loop body in the original program, and so, each block contains a subset of the DDM threads in the program. This reduces the demand on the TSU resources as only a subset of the DDM threads will be executing at a given time. Figure 64 illustrates the dependency graph of a program that is split into three DDM blocks.

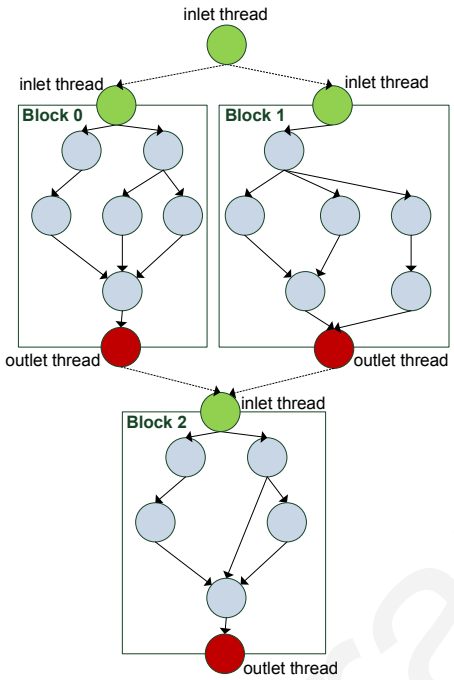


Figure 64: Resource Management - Partitioning a program into DDM blocks

5.6.4 Synchronization Memory Organization

Dynamic Data-Flow execution involves the generation and consumption of tagged data *tokens* [11] in the system. In DDM all the tagged token matching operations are reduced into virtual memory translations and implemented as updates to the Synchronization Memory (SM) structure allocated in main memory. The SM holds the *readycount* values of the different invocations of the DDM threads.

As the operation of the SM is critical for the performance of DDM execution, we have experimented with 3 different implementations:

- **Direct:** Each invocation of a DDM thread is allocated a unique SM entry. The allocation occurs at the time of creating the thread template. Accessing the entry at runtime is a direct operation that uses part of the *context* to index the SM.

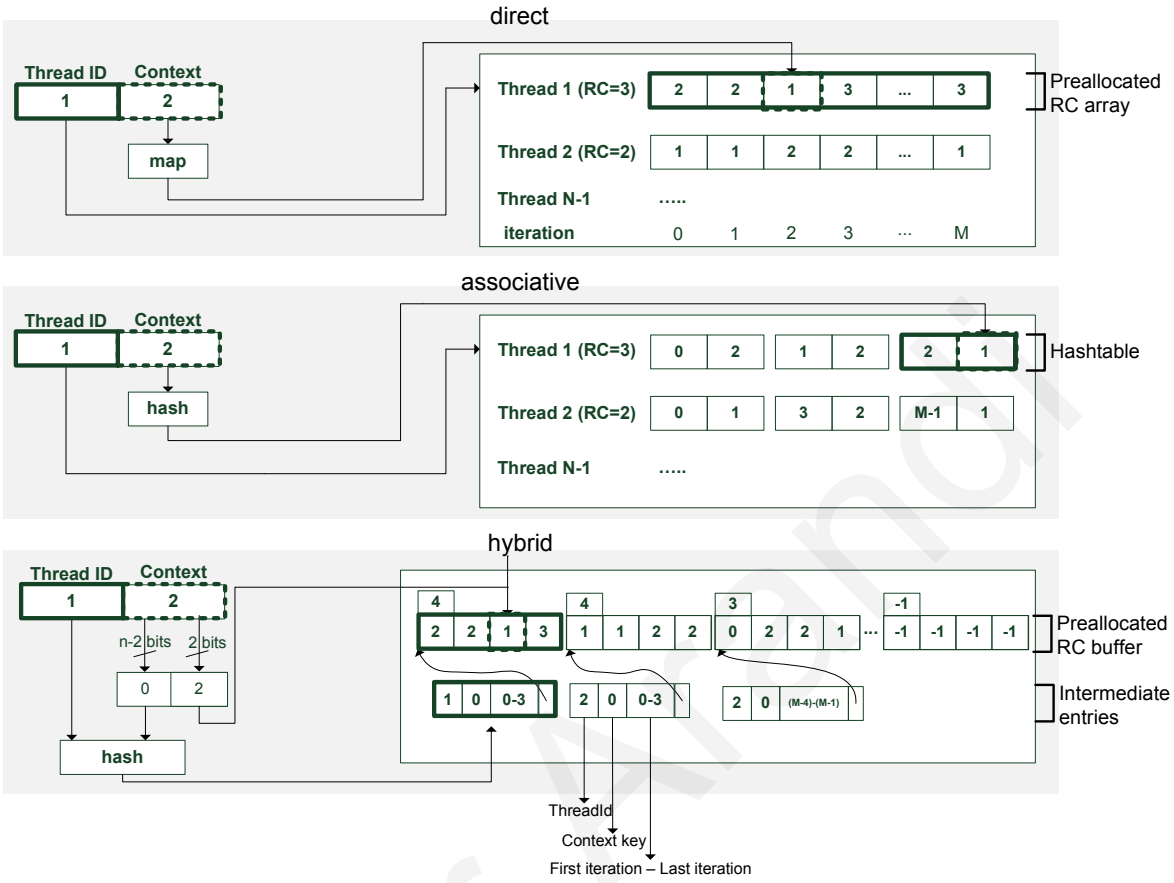


Figure 65: Access Mechanisms in the Three SM Implementations

- **Associative:** A standard hashtable is used to allocate the SM entries. The allocation is performed as the execution proceeds. Accessing the entry is an associative operation.

- **Hybrid:** A pre-allocated buffer is used for holding the SM entries. Allocation and de-allocation within the buffer are performed as execution proceeds. Accessing the entry is performed using an associative operation that uses part of the *context* to locate a list of entries in the buffer, followed by a direct operation using the remaining part of the *context* to index the exact entry.

The *Direct* Implementation

The *direct* implementation involves the least runtime overhead for allocating and accessing the SM entries. However, the programmer is required to provide information on the maximum value the *context* of the thread would reach, which is typically conveyed from loop bounds. This information is conveyed in the *SM_VALUE* parameter of the *DVM_SET_THREAD_TEMPLATE* macro shown in Table 1. This parameter is a bit mask indicating the maximum value the different parts of the *context* would reach (expressed as the closest power of 2 number). For example, a thread implementing a two-level nested loop with the outer loop index ranging from 0 to 100 and the inner loop index from 0 to 250 has a mask value of: **0x007f00ff**. This assumes a 32-bit *context* with the higher 16-bit representing the outer loop index and the lower 16-bit representing the inner loop index. Thus, a 384 entry array is preallocated for holding the SM entries of this thread. The mask is also used at runtime in combination with the thread invocation *context* to access the corresponding SM entry. This method results in redundant allocations if the loop indices don't start from 0 or if the upper bound of the loop is not a power of 2. This can be avoided by providing more information to the TSU on the exact number of entries to allocate and the exact bounds, which would require storing more information and/or a separate interface for the allocation of SM entries. We didn't opt for this as the implemented method was largely adequate for all the applications we ran.

The *Associative* Implementation

The *associative* implementation requires no information from the programmer and has no bound on the size it can reach, but its performance depends on the associative search and the hash function.

The *Hybrid* Implementation

The *hybrid* implementation takes advantage of locality to reduce the size of the SM by allocating a list of entries at a time and most importantly re-using the entries within the preallocated buffer. Increasing the number of entries per list increases the opportunity of benefiting from locality, but at the same time increases the potential of using up the preallocated buffer if the updated invocations are sparse (locality is poor). Each list is associated with a counter indicating the number of valid entries in the list. When the list is allocated the counter is initialized and every time the value of an entry reaches zero the counter is decremented and once it reaches zero the list is marked as free.

Figure 65 depicts the the mechanism to access an SM entry associated with a specific thread invocation (Thread_Id with *context*) in the three implementations. In this example, the *hybrid* implementation utilizes four entries per list.

Supporting Distributed Execution

As mentioned in Section 4.2.1 distributed DDM execution makes the allocation of SM entries more complex since the allocation spans multiple nodes in the system. The allocation of the SM entries is directly influenced by the scheduling policy assigned to the thread. If a thread is assigned the *static* or *modulo* policies, the TSU has enough information to handle the allocation for the three SM implementation. Figure 66-a illustrates the allocation layout for a thread with 200 invocations [0-199] that is assigned the *modulo* policy and the *direct* SM implementation, on a system that has two nodes with 2 cores each. Because the number of nodes, the number of cores within a node, the total range of the invocations and the *modulo* operator are available to the TSU, an efficient allocation scheme is reached.

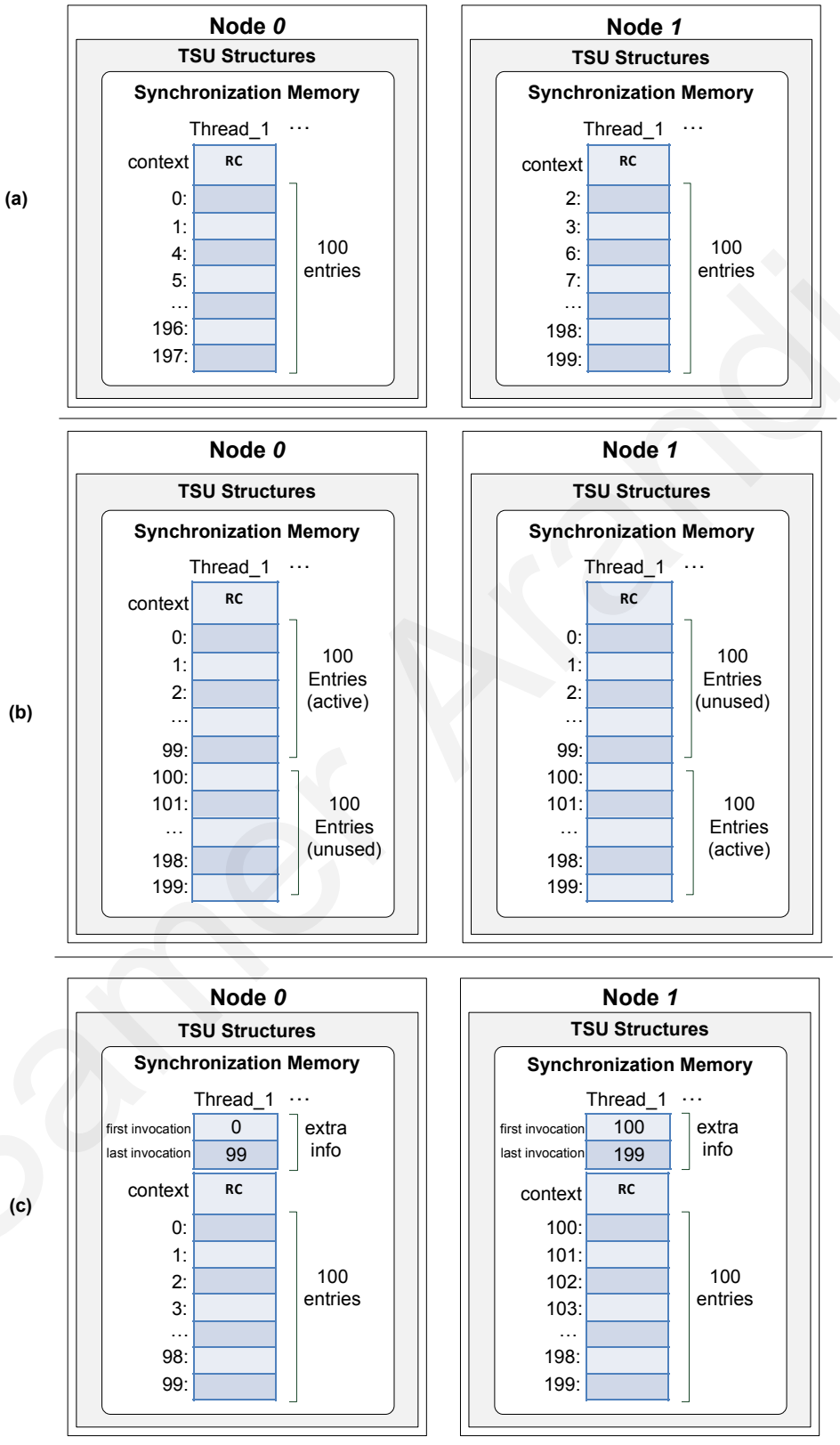


Figure 66: SM Allocation in Distributed DDM Execution

On the other hand, if the thread is assigned a *custom* policy, the three SM implementation need to be re-examined: The *associative* implementation requires no change as the entries will be allocated *on-demand* on the nodes where the corresponding thread invocations are mapped to execute. The *direct* implementations, however, requires extra care as less information is available to the TSU in this case. A simple technique would be to allocate the entire range of the SM entries on each node although only part might be used. This greatly simplifies the DDM-VM program but results in redundant allocations. A more optimized technique provides more information to the TSU on the SM entries to allocate for each node so as to avoid redundant allocations. Figure 66-b & c illustrates the two techniques, respectively. The thread is assigned a *custom* schedule that maps invocations [0-99] to *node 0* and invocations [100-199] to *node 1*. The same reasoning applies to the *hybrid* implementation albeit, with far-less redundant allocations. In the current implementation of the DDM-VM we utilize the first simple technique with *custom* policies as it requires no extra effort from the programmer.

5.7 T-Flux Directives

The 2nd implementation of DDM, the TFlux platform [113], offers a preprocessor tool [119, 114] that allows applications to be easily ported to TFlux by augmenting C code with a set of compiler directives. The preprocessor generates code that contains calls to the TFlux runtime system that can be compiled with commodity compilers.

As a collaborative effort with Andreas Diavastos from the TFlux team, the preprocessor tool was extended to generate code that targets the DDM-VM. The generated code consists of C code augmented with the DDM-VM macros.

We have utilized a subset of the original TFlux directives without modifications and extended another subset to add the information needed for generating the macros. A full description of the utilized TFlux macros is available in Appendix B.

To demonstrate programming with the directives we present the LU decomposition (previously shown in Figure 5.4.5) coded using the extended TFlux directives in Figure 67.

The `ddm kernel` directive specifies utilizing 4 cores for executing the program threads. The `ddm startprogram` and `ddm endprogram` directives specify the boundaries of the program. The `ddm block` and `ddm endblock` directives specify the boundaries of the only DDM block in the program.

The `ddm thread` directive marks the start of the threads and provides part of the meta-data of the thread: The THREAD_ID, the scheduling policy assigned to the thread, the RC and the arity of the thread. It also specifies the thread input/output data. The `ddm endthread` marks the end of the thread and specifies, via the `update` and `cond_update` keywords, the THREAD_ID and *context* of the consumer threads which RC is to be decremented by the TSU. The `@context` keyword and the `@` operator are used to access the value of the *context* and create new *context* values, respectively.

As shown in the figure the code is more compact and clear to read. The program is written in one file including the thread input/output data definitions. The TFlux preprocessor generates one file in the case of the DDM-VM$_s$ and two files in the case of the DDM-VM$_c$ (one file for the part of the program running on the PPE and the other for the part running on the SPEs).

5.8 GCC Auto-Parallelization

The goal of this collaborative effort with Petros Panagyi and other researchers in the DDM group is the automatic generation of DDM-VM code using the GNU Compilation Collection

```
float AA[ROW*COL];
float *A[TILES][TILES]; // TILES = ROW/BS; BLOCK_SIZE=BS*BS*sizeof(float);
int main(int argc, char **argv)
{
  // data initialization
  // runtime initialization

  #pragma ddm kernel 4

  #pragma ddm startprogram

   #pragma ddm block 1

    #pragma ddm thread TID_DIAG kernel(rrobin) readycount 1 arity 1
    import_export(float *T:A[@context][@context]:BLOCK_SIZE)

    diag(T);

    update = @context<TILE-1 ;
    #pragma ddm endthread cond_update(TID_FRONT,@(@context,@context+1):
                                                @(@context, TILES - 1):
                                                 update,
                                      TID_DOWN, @(@context,@context+1):
                                                 @(@context, TILES - 1):
                                                 bUpdate);

    #pragma ddm thread TID_FRONT kernel(rrobin) readycount 2 arity 2
    import(       float *T:A[@context.1][@context.1]:BLOCK_SIZE)
    import_export(float *P:A[@context.1][@context.0]:BLOCK_SIZE)

    front(T,P);

    #pragma ddm endthread update(TID_COMB,@(@context.1,@context.1+1,
                                            @context.0):
                                          @(@context.1,TILES-1,
                                            @context.0);

    #pragma ddm thread TID_DOWN kernel(rrobin) readycount 2 arity 2
    import(       float *T:A[@context.1][@context.1]:BLOCK_SIZE);
    import_export(float *Q:A[@context.0][@context.1]:BLOCK_SIZE);

    down(T,Q);

    #pragma ddm endthread update(TID_COMB,@(@context.1,@context.0,
                                            @context.1+1):
                                          @(@context.1,@context.0,
                                            TILES-1);

    #pragma ddm thread TID_COMB kernel(RROBIN) readycount 3 arity 3
    import(       float *P : A[@context.1][@context.0]:BLOCK_SIZE,
                  float *Q : A[@context.0][@context.2]:BLOCK_SIZE)
    import_export(float *S : A[@context.1][@context.2]:BLOCK_SIZE)


    comb(P,Q,S);


    if (j == @contex.0+1 && @context.2 == @conetxt.0+1) //update DIAG
       update_diag  = 1;
    else if ( @context.1 == @context.0+1) //update FRONT
       update_front = 1;
    else if ( @context.2 == @context.0+1) //update DOWN
       update_down  = 1;
    else //update COMB
       update_comb  = 1;

    #pragma ddm endthread cond_update(TID_DIAG :@(@context.1+1):update_diag,
              TID_FRONT:@(@context.1,@context.0):update_front,
              TID_DOWN :@(@context.2,@context.1):update_down ,
              TID_COMB :@(@context.0+1,@conetxt.1,@context.0):update_comb);

   #pragma ddm endblock

   #pragma ddm update (TID_DIAG : @(0))
   #pragma ddm update (TID_FRONT: @(0,1):@(0,TILES-1))
   #pragma ddm update (TID_DOWN : @(0,1):@(0,TILES-1))
   #pragma ddm update (TID_COMB : @(0,1,1):@(0,TILES-1,TILES-1))

  #pragma ddm endprogram

  // verification, use results here
  }
```

Figure 67: LU Decomposition - Using the extended TFlux directives

(GCC) compiler. In particular, it utilizes the GCC's GRAPHITE (GIMPLE Represented as Polyhedra with Interchangeable Envelopes) [97, 17, 120] framework, which adds high-level loop nest optimizations in GCC, as the core analysis and transformation engine that is leveraged to implement a DDM-centric parallelization pass. This pass takes advantage of the existing GCC infrastructure and adds the necessary support for generating DDM threads with the appropriate calls to the DDM-VM. We provide a brief description of the GRAPHITE pass in GCC, before presenting the work on generating DDM code using GCC.

The GRAPHITE Infrastructure

The GRAPHITE pass in GCC performs the following tasks:

- Extracts the polyhedral model representation out of the GCC three-address GIMPLE representation. This is performed in two stages:
    1. The Static Control Parts (SCoPs) are first outlined from the control-flow graph. SCoPs are Single-Entry-Single-Exit regions of the control-flow graphs. The only memory references that are allowed within SCoPs are affine accesses on arrays.
    2. The polyhedral representation is then constructed for each SCoP. The polyhedral information is attached to each basic block in a SCoP and consists of iteration domain, schedule and data access functions. All these information is represented as systems of affine equalities and inequalities that can be easily represented and manipulated.
- Performs various data dependence analyses, transformations and optimizations on the polyhedral model. This includes transformations like loop interchange, strip-mining, distribution and blocking.

- Generates the GIMPLE three-address code resulting from the applied transformations. The iteration domains are converted into loops and conditionals, and data references are converted into actual scalar and array accesses with all necessary address computations.

Implementing DDM support in GCC

The key to DDM code generation is the identification of DDM threads and the dependences amongst the DDM thread invocations. To this end, the GRAPHITE framework is used as the core analysis and transformation engine. A new GRAPHITE-to-GIMPLE translation scheme was introduced to put the reconstructed SCoP in a DDM-compliant form.

The translation scheme is split into four major stages:

1. Reconstruction of imperative control structures and the recording of detailed information about each created loop. The information is needed to convert a loop nest into a DDM thread and to generate the necessary context manipulation and thread activation primitives.

2. Identification of the loops that should be converted into DDM threads. This was the most complex task at hand. The utilized criteria depend on the properties of target architecture: The amount of available storage, the size of the internal TSU queues and the optimal number of concurrently active DDM invocations, etc.

3. Outlining of the qualified outermost loop(s) to a worker function and the insertion of the corresponding DDM-VM runtime calls around the outlining point(s). The outlining stage adapted the mechanism initially developed for OpenMP to be used for DDM.

4. Refinement of the outlined function to expose further threads.

The DDM code generation utilizes a scheme where all the DDM threads are outlined into one worker function. This function consists of an initialization preamble and an infinite loop that

```
int threads_main (int coreid)
{
     unsigned int raw_context;
     unsigned int thread_id;
     Init_RUNTIME(coreid);

     while (1) {
        Check_Fin (coreid);//ensures the TSU is notified of termination
        GetNextThread (&thread_id, &raw_context, coreid);
             switch (thread_id) {
                     case THREAD_1 :
                             // decode context
                             // execute thread body
                             break;
                     case THREAD_2 :
                             // decode context
                             // execute thread body
                             break;
                     ...
                     default:
                             return thread_id;
             }
     }
}
```

Figure 68: The structure of the generated worker function

repeatedly requests ready threads information from the TSU. This request blocks until the TSU returns the thread identifier and *context* of the ready thread. Following that one of the branches of a switch statement is selected based on the thread identifier and the *context* components are retrieved before the corresponding thread body is executed. Figure 68 shows the structure of the worker function.

All the extracted thread *meta-data* are loaded at the site of the first outlining, which ensures that the dependency graph has been completely loaded into the TSU before starting the execution of the graph.

Preliminary Performance Evaluation

When comparing the performance of the sequential code with the parallel DDM-VM generated code for the blocked matrix multiplication on an 8 core machine, the parallel code outperformed the sequential by a factor of 2.4x. This result is very encouraging especially that it was achieved without tuning and in the presence of overheads caused by redundant control structures. A detailed description of this on-going effort is presented in [91].

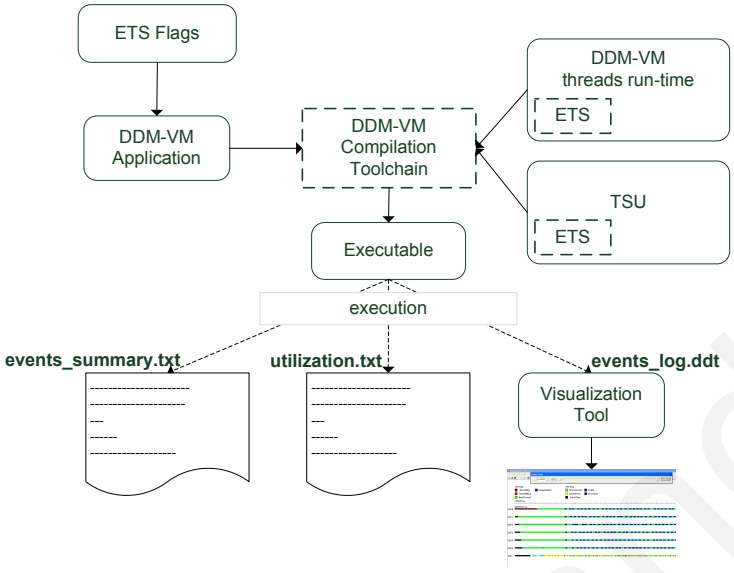


Figure 69: The Event Tracing System (ETS)

5.9 Monitoring and Visualization Tools

The development and debugging of concurrent programs is a difficult and complex task. The DDM-VM provides monitoring and visualization tools that make this task easier. In this section we describe the Event Tracing System (ETS), which is the part of the DDM-VM encompassing these tools.

The ETS records the most important events occurring during the execution of the DDM-VM application. The events include TSU execution events and DDM Threads execution events. The ETS also uses the collected information to provide statistics regarding the usage and utilization of the different TSU structures. All this information provide an accurate understanding of the system performance and execution details and allow the programmers to optimize the developed DDM-VM applications.

The code of the DDM-VM is instrumented with a number of function calls implementing the functionality of the ETS. The calls are enabled when linking with the debug version of the DDM-VM runtime library. After the DDM-VM application finishes execution, the collected events and

statistics are processed and recorded in multiple log files. The first log file (*events_log.ddt*) is used as input to the *Visualization Tool* that displays the execution events in a time-line fashion. This provides the programmer with a detailed account of the execution of DDM-VM applications and help the programmer to optimize the application easily.

The other two log files report a summary of the execution events (*events_summary.txt*) and the utilization of the TSU structures and other statistics (*utilization.txt*). The provided information gives insight into the performance of the different applications and the inner work of the TSU.

Activating the Event Tracing System introduces inevitable overheads to the execution of the DDM-VM application. However, the ETS is implemented efficiently so as to minimize such overheads. In addition, when linking to the release version of the runtime library the ETS system is not activated and no overhead whatsoever is suffered. Furthermore, the user controls the activation of specific parts of the ETS using a number of ETS flags passed as parameters when initializing the runtime. This helps focus the tracing to the relevant parts and further reduce the overheads. Figure 69 illustrates an overview of the ETS system. The ETS also supports monitoring and visualizing the events of distributed execution. Figure 70 depicts the Visualization Tool displaying the events of the execution on 3 nodes.

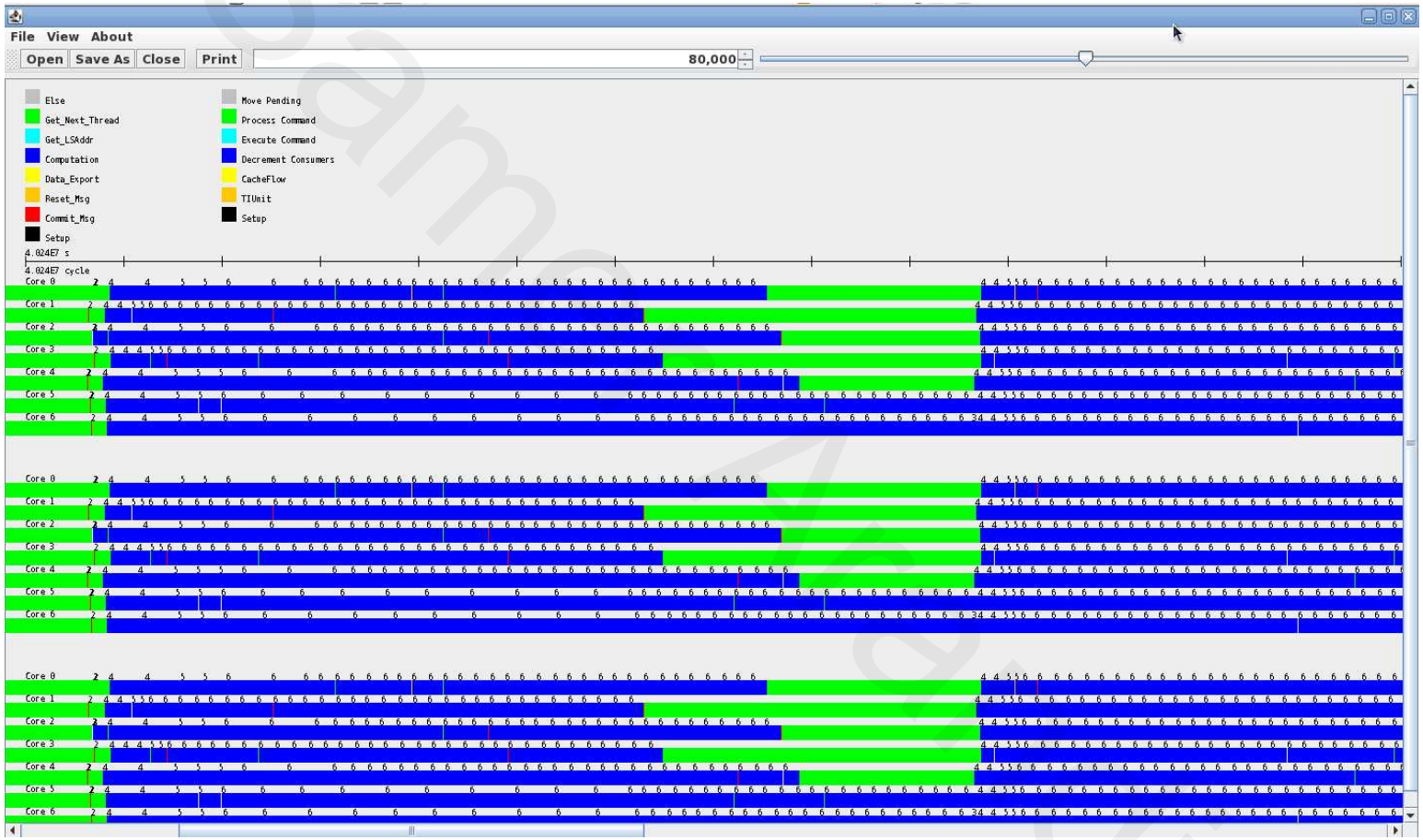


Figure 70: Visualization Tool Screenshot - Distributed DDM Execution

Please refer to Appendix C for a detailed description of the implementation and optimization details of the ETS system, in addition to the format of the various generated statistics and log files.

Chapter 6

Runtime Dependency Resolution

6.1 Introduction

A large class of programs exhibit a computational pattern in which the producer-consumer dependencies cannot be determined at compile-time. This typically occurs in programs that utilize pointers and so the addresses of the produced/consumed data are only determined at run-time.

We propose to a technique for representing this class of programs in DDM that makes use of I-Structures [15] for handling the synchronization between the producer and consumer threads in a *split-phase* manner, i.e. a request issued to an I-structure is independent in time from the received response. Utilizing this technique we combine the efficiency of compile-time dependency resolution with the flexibility of run-time dependency resolution.

6.2 I-Structures

An I-Structure [15] is a type of storage controller that obeys the single-assignment rule: Each element is written only once but can be read multiple times. If a read request arrives for a storage element that has not been written yet, the controller defers the read until a write arrives [10]. This

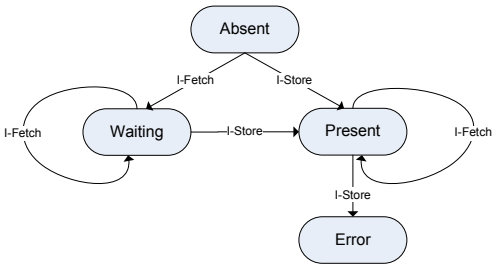


Figure 71: State Transitions for I-Structure Elements

property of I-Structures provides the synchronization needed for exploiting producer-consumer parallelism without the risk of read-write races [10]. We use the same property to discover the producer-consumer dependencies at runtime.

The basic idea in I-Structures is to add status bits to the storage cells in addition to a queue for holding deferred reads. The status of each element or storage cell of the I-structure can be:

- present: the cell data is valid and can be freely read but any attempt to write to it will be considered an error.
- absent: nothing has been written into the cell yet and no attempt has been made to read it. A write operation is allowed.
- waiting: nothing has been written into the cell yet, but at least one read request was attempted (deferred read). When this cell is written all the deferred reads must be satisfied.

A read operations on an I-Structure is commonly referred to as an *I-Fetch* and a write operation as an *I-Store* [15]. Figure 71 illustrates the state transitions of the I-Structure cells.

6.3 Run-time Dependency Resolution with I-Structures

Assuming a program has been partitioned into a number of DDM threads. A problem arises when part or all of the threads perform read operation(s) on data items whose address is resolved at

runtime. Although, other threads in the program potentially produce such data items, because the address of the consumed data is only determined at runtime, establishing the producer-consumer relationships amongst the threads is not possible at compile-time. To solve this problem we propose the following technique:

- For every thread t that performs at least one read operation on data items which address is resolved at run-time a *proxy* thread t' is introduced. Thread t' replaces t in all the Consumer Lists of its *explicit* producer threads, i.e., threads that are identified as its producers at compile-time.

- The RC of t' is set to the number of *explicit* producers of t. The RC of t is set to the number of read operations performed on data items with run-time resolved addresses.

- For every such read a special *I-Fetch* request is issued by t'. The first parameter of the *I-Fetch* is the address of the data to read, the second and third parameters are the thread identifier and *context* of t. When the *I-Fetch* request is executed at runtime, it checks the I-Structure for the data address, if the address exists, i.e. the data has been produced, a request is sent to the TSU to decrement the RC of thread t. If the data was not produced yet, the request is added into a pending list inside the I-Structure. Note that the *I-Fetch* in this manner is faithful to the non-blocking property of DDM, i.e. it doesn't cause a wait by the issuing thread.

- For every thread producing data that is potentially read by t, a special *I-Store* request is issued when the thread finishes execution. The *I-Store* registers the address of the produced data in the I-Structure, which results in sending all the existing pending requests on that address to the TSU.

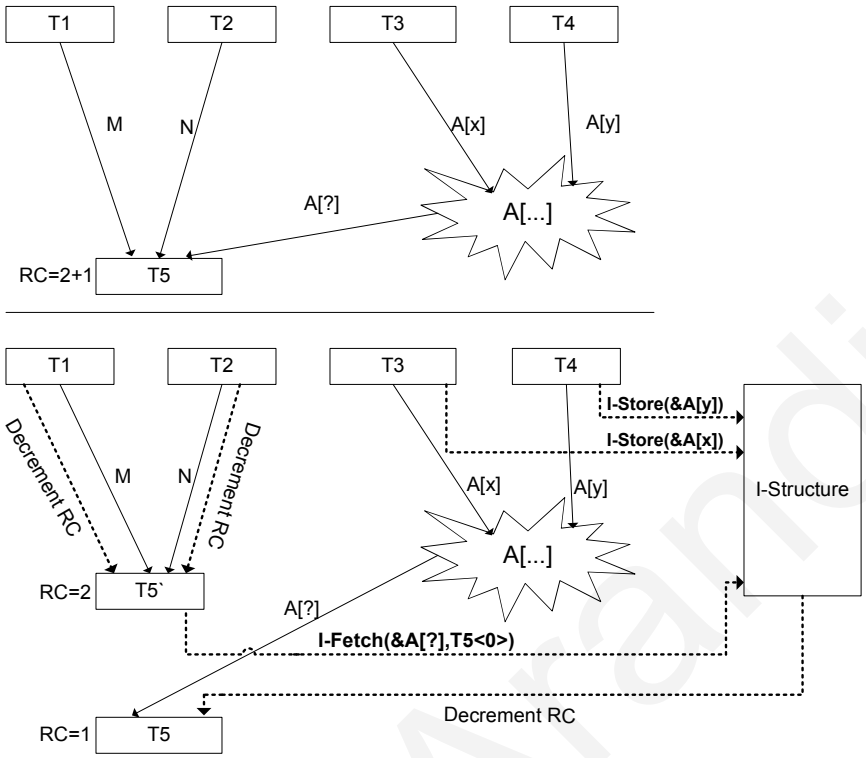


Figure 72: DDM-VM Program with Run-time Determined Dependencies

It is important to note that in contrast with the traditional usage of I-Structures, we only keep the addresses of the data in the I-Structure storage cells and the data itself resides in the conventional memory. We explain this technique further with the help of an example.

6.4 Example

Figure 72 illustrates a synthetic example of a simple DDM-VM program composed of five threads. In the upper part of the figure thread *T5* consumes data items M and N produced by threads *T1* and *T2*, respectively. In addition, *T5* consumes an element of the array *A*, however, the exact address of the element (its array index) is determined at runtime. Array *A* elements are produced by threads *T3* and *T4* and so it is only at runtime that the producer-consumer link between one of the two threads and *T5* is established. The problem is to ensure that *T5* executes

Table 2: DDM-VM I-Structure Macros

DDM-VM I-Structure Macros	
`DVM_IFETCH(ADDR,THREAD_ID,CONTEXT)`	Checks if ADDR exists in the I-Structure. If so a request to decrement the RC of the invocation with thread identifier THREAD_ID and *context*=CONTEXT is inserted in the AQ, otherwise the request is added to a pending list.
`DVM_ISTORE(ADDR)`	Registers ADDR in the I-Structure. Any pending requests on this address are served.
`DVM_ISTRCUT_INIT(SIZE,CORE_NUM)`	Initializes the I-Structure and sets the initial size of the buckets and the number of cores the will access the structure.
`DVM_ISTRUCT_SHUTDOWN()`	Shuts down the structure and perform cleanup tasks

only after the array element it requires has been produced by either *T3* or *T4*. The *"?"* symbol in the figure is used to indicate a value that is determined at runtime.

The lower part of the figure shows how this problem is solved using the proposed technique. A new *proxy* thread *T5′* is introduced and its RC is set to two. T5′ replaces T5 in the Consumer Lists of T1 and T2. An *I-Store* operations is added at the end of *T3* and *T4* to register the address of the produced element of array *A*. Moreover, an *I-Fetch* operation is added to *T5′*. When *T1* and *T2* finish execution, they notify the TSU, which decrements the RC of *T5′* twice making it zero. Consequently, *T5′* executes, it evaluates the address of the requested element in A and issues an *I-Fetch* on that address. If the address is found in the I-Structure (an I-Store with the address of that element was executed previously by *T3* or *T4*) a request to decrement the RC of *T5* is immediately sent to the TSU and once processed, *T5* RC becomes zero and it runs. If the address was not found, the request is enqueued in a pending list waiting to be sent to the TSU once the corresponding *I-Store* operation occurs. Table 2 lists the two DDM-VM macros implementing the I-Fetch and I-Store operations in addition to the macros needed for initializing and cleaning up the I-Structure.

6.5 The I-Structure Implementation

The efficiency of the I-Structure implementation is central to this technique. In particular, the operation of finding the entry corresponding to an arbitrary address, which incurs a non-trivial

overhead. One solution is to implement this search operation as a *hashmap* search. A similar approach was used to implement the search in the I-Structure software cache described in [134]. This solution is a general one that can handle any type of accessed data (e.g. scalars, arrays, lists, etc.) including recursive and dynamically allocated data, as each I-Structure entry is associated with an arbitrary address. On the other hand, if the accessed data consists of arrays of a pre-determined size, it is possible to utilize a different I-Structure organization consisting of an array of preallocated entries, each corresponding to one array element. This removes the overhead of the associative search on the expense of extra memory storage. In this work we explore the first approach as it is more general.

The low-level design of the software I-Structure depends on the implementation of the DDM-VM. In the DDM-VM$_c$, due to the limited size of the Local Store (LS) memory on the SPE cores executing the threads and the sharing of the same I-Structure across all the SPEs, the I-Structure is allocated in main memory. Consequently the *I-Fetch* and *I-Store* operations are executed by the general purpose PPE core running the TSU and so there is no concurrent access to the I-Structure hashmap, which makes the design simpler. In the case of the DDM-VM$_s$, however, the *I-Fetch* and *I-Store* are performed by the runtime threads and so concurrent access to the hashmap occurs. Although this complicates the design, it also permits a distributed implementation of the search operation on the hashmap.

6.6 Hopscotch Hashing algorithm

To implement the I-Structure we used the Hopscotch Hashing algorithm [52] because it is highly-scalable, it outperforms existing hashing algorithms on both single-core and multi-core machines and most importantly it delivers good performance even when the hashmap is more than 90% full. This particular feature is very important in the case of I-Structures, as the addresses of

produced data are typically kept throughout the duration of the application execution, thus increasing the density of the hashmap. We have leveraged this hashmap in our I-Structure implementation and used the data address as the key to the stored hashmap entries. The entries have a presence field (performing a similar function to the presence bit of a traditional I-Structure) and a head pointer that holds a list of pending requests on the data address.

6.7 Discussion

The proposed technique has two shortcomings: (i) the inevitable overheads of the I-Structure operations existing despite the optimized hashing algorithm (ii) and the fact that *proxy* threads must execute the part of the original thread code that evaluates the address of the accessed data, as this address is needed for the I-Fetch. Because this code is also executed by the original thread, an issue arises if the code is expensive to execute twice or altogether impossible (for example if it calls a random number generator).

To handle the first issue we employ an optimization that is applicable for the class of programs in which it is possible to know, by analyzing the program, how many times an I-Fetch will refer to a certain data address. For such cases we can assign a counter-value to the I-Structure entry associated with that address. The counter is decremented for every I-Fetch operation on that entry and once the counter reaches zero the entry is removed. This reduces the total number of hashmap entries, consequently improving the search performance and reducing the overheads. The counter value is assigned via the same I-Store registering the data address in the I-Structure.

To handle the second issue, we allow the code to be executed twice as long as it is possible and inexpensive. Otherwise, we execute it once at the *proxy* thread and channel any results required by the original thread as input data produced by the proxy thread. A combination of compilation analysis and profiling can be used to select the appropriate approach for each program.

The proposed technique has many advantages. Not only does this technique allow handling programs with run-time dependencies, but it does so while allowing compile-time dependencies to be utilized at the same time. Thus, obtaining the benefits of both approaches.

Further, the technique is very beneficial for compilation techniques that target the generation of data-flow code. Traditionally, when the compiler is unable to uncover the dependencies due to the existence of pointers, for example, it falls back to running the code sequentially. However, leveraging this technique, the compiler can fall back to generating parallel code and leave the discovering of dependencies to occur at runtime. Regardless of the involved overheads this alternative is expected to yield -in many cases- substantially better performance than running the code sequentially.

Finally, this technique improves the programmability of the DDM-VM. In the case of complex code with complicated dependencies, the programmer can utilize this technique to quickly develop and run the DDM-VM application without going through the most involving step, which is the dependency analysis. Dependency analysis can be incorporated later to encode the dependencies at compile-time for improving the performance. In fact, we envision an extension of this technique that records the discovered dependencies at runtime so as to provide feedback for guiding the uncovering of the dependencies and encoding them in the program.

Chapter 7

Evaluation

7.1 Introduction

In this Chapter we present the evaluation results for the two DDM-VM implementations for both single-node execution and distributed/multi-node execution. We present the evaluation of the DDM-VM$_c$, followed by the evaluation of the DDM-VM$_s$ and conclude this chapter with a summary of the findings.

7.2 The DDM-VM$_c$ Evaluation

In this section we present the evaluation of the DDM-VM$_c$. The first part of this section evaluates the effect of the Resource Management, Synchronization Memory Organization and the Locality Exploitation on the performance using the MatMult and Cholesky benchmarks as case studies. The second part presents a comprehensive performance evaluation using all the benchmarks, which also includes a comparison with state-of-the-art systems targeting the Cell. In the third part we present the evaluation of the distributed DDM-VM$_c$ execution.

7.2.1 Experimental Setup

The DDM-VM$_c$ runs on a Sony Playstation 3 (PS3) machine with Linux 2.6.23-r1 SMP OS and the IBM Cell SDK version 2.1. The Cell processor powering the PS3 has 6 SPEs available for the programmer out of the original 8 (one is reserved for the *hypervisor* and one is shutdown to increase the yield). The cores run at 3200 MHz and have access to 256 MB of RAM. For the evaluation of the distributed execution we used a cluster of four PS3 machines. The machines were connected using an off-the-shelf Giga-bit Ethernet switch with a latency of approximately 250 μs.

The benchmark suite used in the evaluation consists of ten applications featuring kernels widely used in scientific and image processing applications. The characteristics of the benchmarks are presented in Table 3. For the benchmarks working on matrices, the matrices are dense single-precision floating-point, except for the IDCT benchmark, which works on short integers.

All of the benchmarks were coded in C using the DDM-VM *macros* and compiled by the compilers available from the IBM Cell SDK V2.1. All reported speedup results are relative to the DDM execution time on one SPE core.

Table 3: The benchmarks suite characteristics - DDM-VM$_c$

Benchmark	Description	Average Granularity of Benchmark Threads		Problems Size				
		Granularity	Execution Time	Small	Medium	Large	XLarge	XXLarge
MatMult	Blocked Matrix Multiplication	64x64 block	22.1μs	512x512	1024x1024	2048x2048	3072x3072	-
Cholesky	Blocked Cholesky Factorization (vectorized)	64x64 block	22μs	512x512	1024x1024	2048x2048	3072x3072	-
Cholesky-S	Blocked Cholesky Factorization (scalar)	64x64 block	8.2ms	512x512	1024x1024	2048x2048	3072x3072	-
LU	Blocked LU Decomposition	64x64 block	1.82ms	512x512	1024x1024	2048x2048	3072x3072	-
FDTD	2D Finite Difference Time Domain [133]	304 Y-Cells	28.65μs	304x304	608x608	1216x1216	-	-
		608 Y-Cells	58μs					
		1216 Y-Cells	116μs					
RK4	4th order Runge-Kutta (ODE solver)	variable	variable	512K	2K	3K	-	-
Conv2D	9x9 convolution filter	32x32 block	12.28μs	512x512	1024x1024	2048x2048	3072x3072	4096x4096
		64x64 block	48.11μs					
		96x96 block	107μs					
IDCT	Inverse Discrete Cosine Transform	32x16 block	12.37μs	512x512	1024x1024	2048x2048	3072x3072	4096x4096
		64x32 block	49.21μs					
		64x64 block	98.8μs					
Trapez	Trapezoidal rule for integration	variable	variable	168K steps	337K steps	675K steps	5400K steps	10800K steps
MatAdd	Matrix Addition	64x64 block	4.6μs	256 iteration	1024 iteration	4096 iteration	-	-
MatCopy	Matrix Copy	64x64 block	4.6μs	256 iteration	1024 iteration	4096 iteration	-	-

7.2.2 Optimizations Evaluation

In this section we evaluate the effect of the factors discussed in Section 5.6 on the performance of DDM-VM programs. We use the MatMult and Cholesky benchmarks as case studies. The first application is a representative of applications with a simple dependency graph and the second is a representative of applications with a complex dependency graph. Moreover, both applications are computationally intensive and performance-sensitive. The result of this evaluation is used to guide the performance optimization for all the benchmarks in the rest of the evaluation sections.

7.2.2.1 Resource Management

To assess the DDM-VM resource management control mechanisms we have executed two sets of experiments for both benchmarks. In the first, we have varied the size of the Extended Firing Queue (ExFQ) and in the second, we have utilized Loop Throttling and varied the limit on the number of concurrent invocations of the throttled threads. To neutralize the effect of the ExFQ on the second set we have chosen a relatively large size for the ExFQ (ExFQ=6). Figure 73 depicts the results.

In the first set of experiments, the results show that for both applications as the size of the ExFQ increases the concurrency increases and the performance improves reaching its best when the size is 3. The reason for this particular size is that the space allocated for the DDM Cache on the LS of each SPE can fit at maximum the data of 3 concurrent invocations of the most computationally intensive threads of the two applications. When the size increases beyond 3 the surplus concurrency causes the performance to degrade. In the second set of experiments utilizing loop throttling, a similar effect to the one in the first set is observed. The effect of throttling on Cholesky is smaller in comparison to MatMult as only one out of the five threads in Cholesky was throttled.

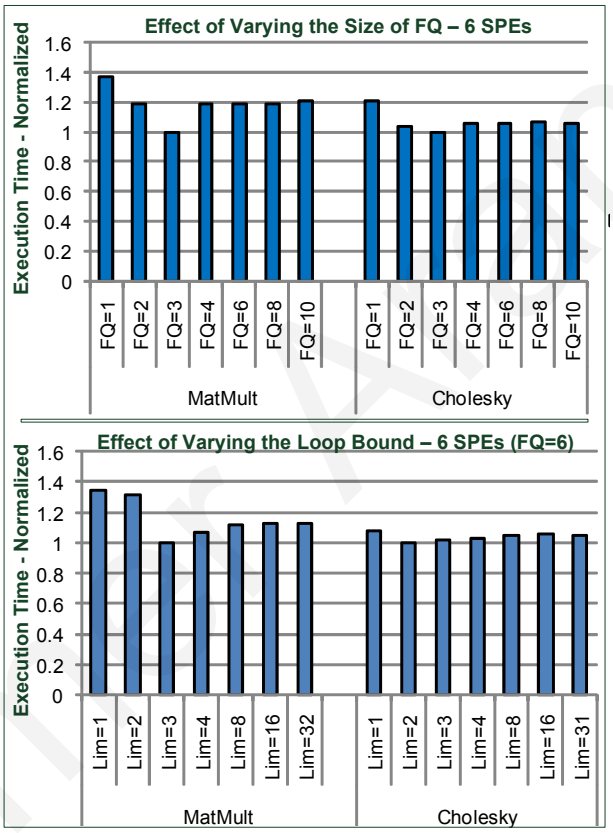


Figure 73: Resource management control - Effect of Firing Queue (ExFQ) size and Loop Throttling on performance

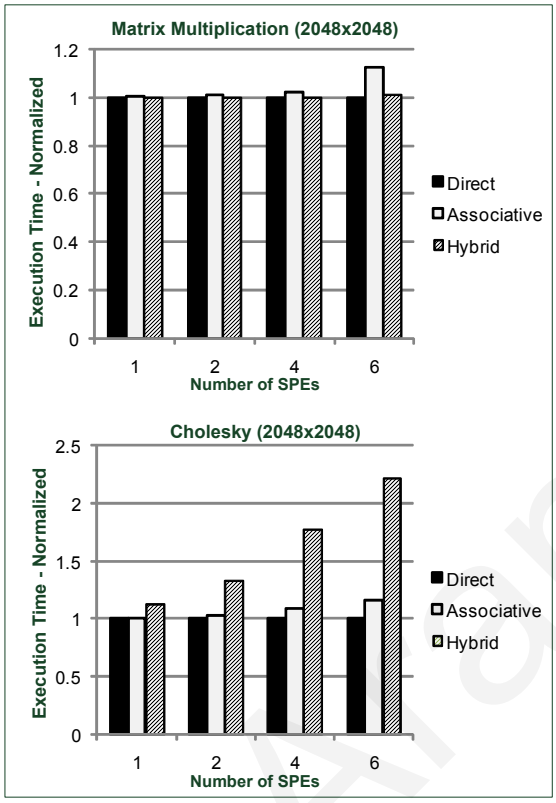


Figure 74: Effect of the different Synchronization Memory implementations on performance

TSU resource control mechanisms like setting the size of the ExFQ has a global effect that applies to all the threads in the program, while loop throttling can be used to control individual threads for fine tuning the performance.

7.2.2.2 Synchronization Memory (SM) Organization

To study the effect of the Synchronization Memory implementations on the performance, we executed the two applications under the 3 different implementations. The results are illustrated at Figure 74.

As expected the *direct* implementation achieves the best performance for both applications as it incurs the minimum overhead for updating the SM entries. The *associative* implementation performs 2nd best on average. The overhead of the associative updates in this implementation

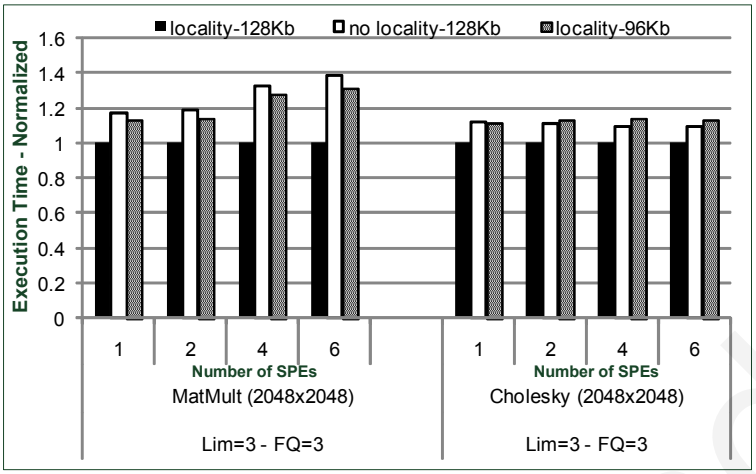


Figure 75: Effect of locality on performance

increases when the number of cores is high as the TSU is working more in that case. The *hybrid* implementation performs very close to the *direct* and better than the *associative* for MatMult, but performs less than the two other implementations for Cholesky. In MatMult the execution of the threads proceeds consecutively generating regular patterns of updates to the SM which is captured well by the re-use mechanism of *hybrid*. The Cholesky application has a much more irregular pattern of execution which generates non-consecutive updates that cannot be captured well. This results in more allocations and more associative searches that degrade the performance. One possible improvement to the *hybrid* implementation is to utilize information on the expected pattern of the threads execution to guide the re-use of entries.

7.2.2.3 Locality and Data Re-use Exploitation

To study the effect of exploiting locality, we executed the two applications with and without locality. Figure 75 illustrates the results. The black and white bars depict the normalized execution time for both cases. The improvement in performance when exploiting locality was achieved by simply identifying the threads that can benefit from locality and adding the *DATA_KEEP* and *DATA_REUSE* flags to their macros and the rest was automated by S-CacheFlow.

It is worthwhile to note that the main source of improvement is the reduced demand of the LS space. Enabling locality for the MatMult, allows the data of 3 invocations of the thread performing the multiplication to fit concurrently in the DDM cache, since one of the input blocks is re-used by all 3 invocations. When locality is not enabled, the data of 2 invocations only can fit. Fitting the data of more threads allows the TSU better chance to prefetch data and overlap latencies with computation which improves the performance. The Cholesky application benefits similarly but to a lesser degree as only one of the computational threads of the application can benefit from re-use.

To confirm our analysis we have executed both applications with locality enabled after reducing the size of the DDM Cache from 128KB to 96KB which has a similar effect on the number of threads that can fit its data concurrently. The results represented by the shaded bar show that the performance degrade in a fashion corresponding to the case when no locality is enabled.

The results demonstrate the deep implications the size of the LS memory has on the execution behavior and consequently the importance of taking into account the size of the working set when choosing the granularity of the threads.

7.2.3 General Performance Evaluation

In this section we present a comprehensive performance evaluation using all the benchmarks. For all the benchmarks we have used the *direct* SM technique, enabled locality and used the combination of *ExFQ* size and throttling limit value that produced the best performance.

We compare the two implementations of S-CacheFlow and study the effect of thread granularity and input size on performance. We demonstrate the potential of the DDM-VM$_c$ in exploiting concurrency and tolerating latencies and compare its performance with state-of-the-art systems. Finally, we present the evaluation of distributed DDM-VM$_c$ execution.

7.2.3.1 Thread Granularity and S-CacheFlow Implementations

To assess the effect of thread granularity and the two S-CacheFlow implementations on performance we executed the benchmarks under both implementations. Note that different benchmarks have different thread granularities and for some of the benchmarks we have executed the same benchmark with varying thread granularities. Table 3 reports this information for every benchmark. The speedup results are depicted in Figure 76. The baseline for the speedup is the best execution out of the two implementations on one SPE.

Thread Granularities

The results show that the performance improves as the granularity increases. This is expected, as higher granularities amortize better the scheduling overheads of the TSU and S-CacheFlow operations and allow DDM-VM$_c$ to hide the latency of data transfers through prefetching/multi-buffering. Applications with small granularity do not scale when the number of SPEs increases to four and higher as the TSU is doing more work then and the computation is not sufficient to totally overlap the TSU work. However, when the thread granularity is increased (for example using a larger block size) the applications scale almost linearly. In certain cases, increasing thread granularities is bounded by the limited size of the LS, hence applications like MatAdd and MatCopy, which have a poor computation/data ratio, cannot benefit from increasing the granularity as this requires larger blocks that don't fit.

S-CacheFlow vs. Distributed S-CacheFlow

Comparing the results of the two S-CacheFlow implementations, the distributed S-CacheFlow, in general, performs as well as or better than the basic S-CacheFlow on all of the benchmarks. The advantage of the distributed implementation is clear when the number of cores increases to 4 and

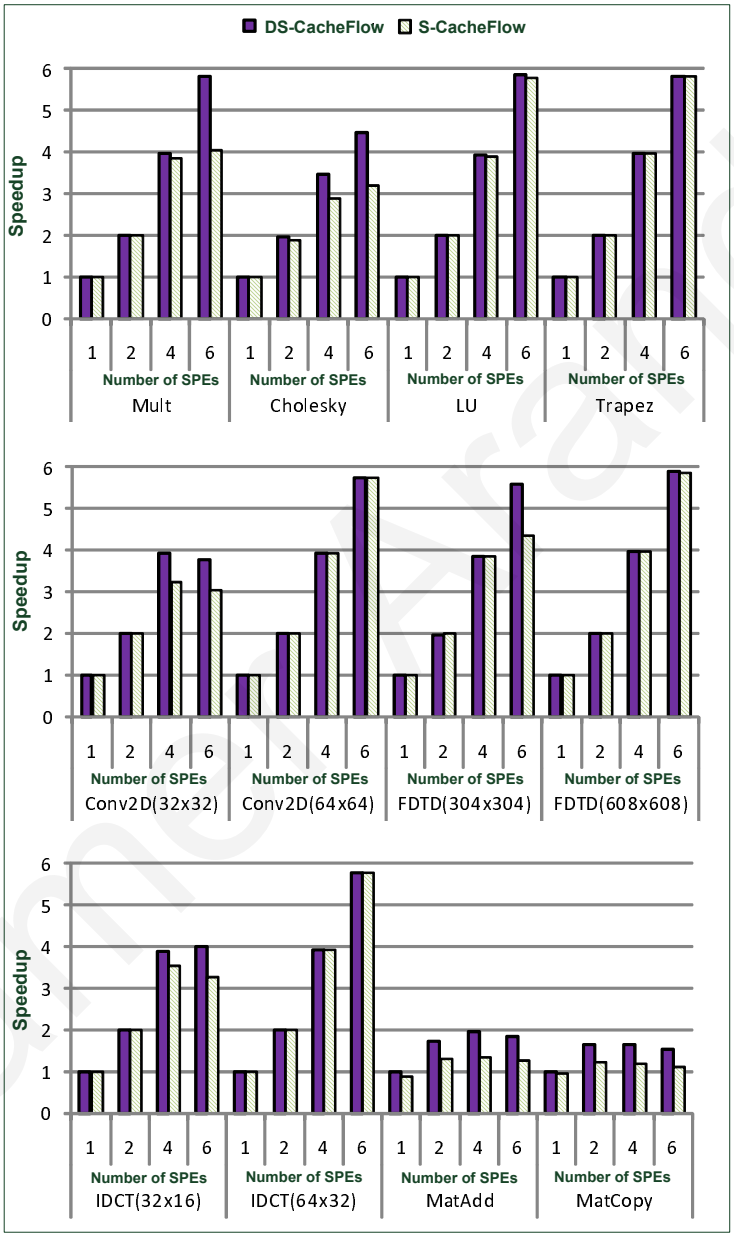


Figure 76: Effect of thread granularity and S-CacheFlow vs. Distributed S-CacheFlow

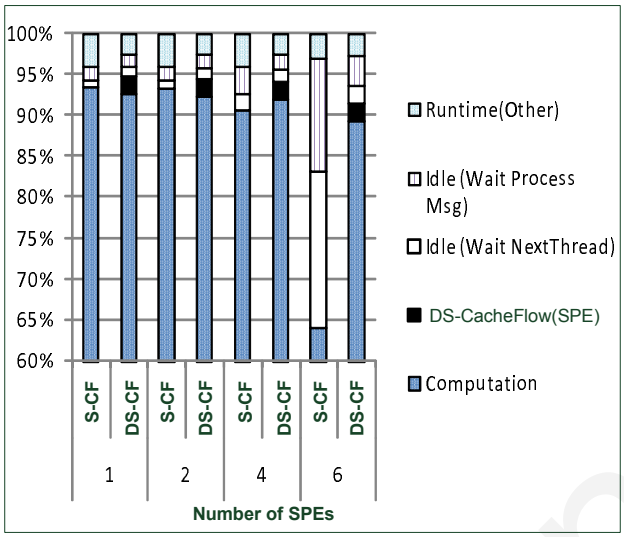


Figure 77: S-CacheFlow vs. Distributed S-CacheFlow - MatMult SPE runtime execution activities

higher, as previously explained in section 4. It is worthy to note that both implementations perform equally well for benchmarks that are not data-intensive (Trapez) or for the ones that have a large enough granularity (e.g. LU) that allows the TSU to overlap the work of scheduling and data management at higher number of cores.

Figure 77 depicts the average activities of the SPEs for the execution of MatMult under the two S-CacheFlow implementations. For clarity we show only the upper 40% of the graph since all the SPEs had average utilization higher than 60%. The results show that up to 4 SPEs, the SPEs spend more than 90% on computational work. At six SPEs -however- the utilization drops to 64% for the basic S-CacheF-low because the PPE becomes a bottleneck due to the demand of the S-CacheFlow. The distributed implementation does not suffer from this and the time spent executing the computational load is kept around 90%. As such, the distributed S-CacheFlow has been adopted as the default for the DDM-VM$_c$.

Figure 78: DDM-VM$_c$ latency tolerance

7.2.3.2 Concurrency and Latency Tolerance

To evaluate the potential of DDM-VM$_c$ in exploiting concurrency and tolerating synchronization and memory latencies, we have performed a number of experiments in which we limit the number of threads that can be scheduled concurrently to 1 (purely sequential scheduling of DDM-VM$_c$ applications), 2 and 3. We compare the results with a normal (non-DDM) sequential program. Figure 78 depicts the results for five of our benchmarks.

The results show that when the limit is set to one (DDM-1) the TSU overhead is simply added to the critical path. When the limit increases to 2 the performance improves as the TSU is able to overlap the overhead of scheduling one thread with the execution of another. When the limit is set to 3 the execution finishes in time less than the sequential, as the automatic prefetching takes effect and, further, overlaps the latency of data transfers with the execution. The results illustrate that DDM-VM$_c$ effectively leverages the decoupling of synchronization and computation for maximum tolerance of latencies.

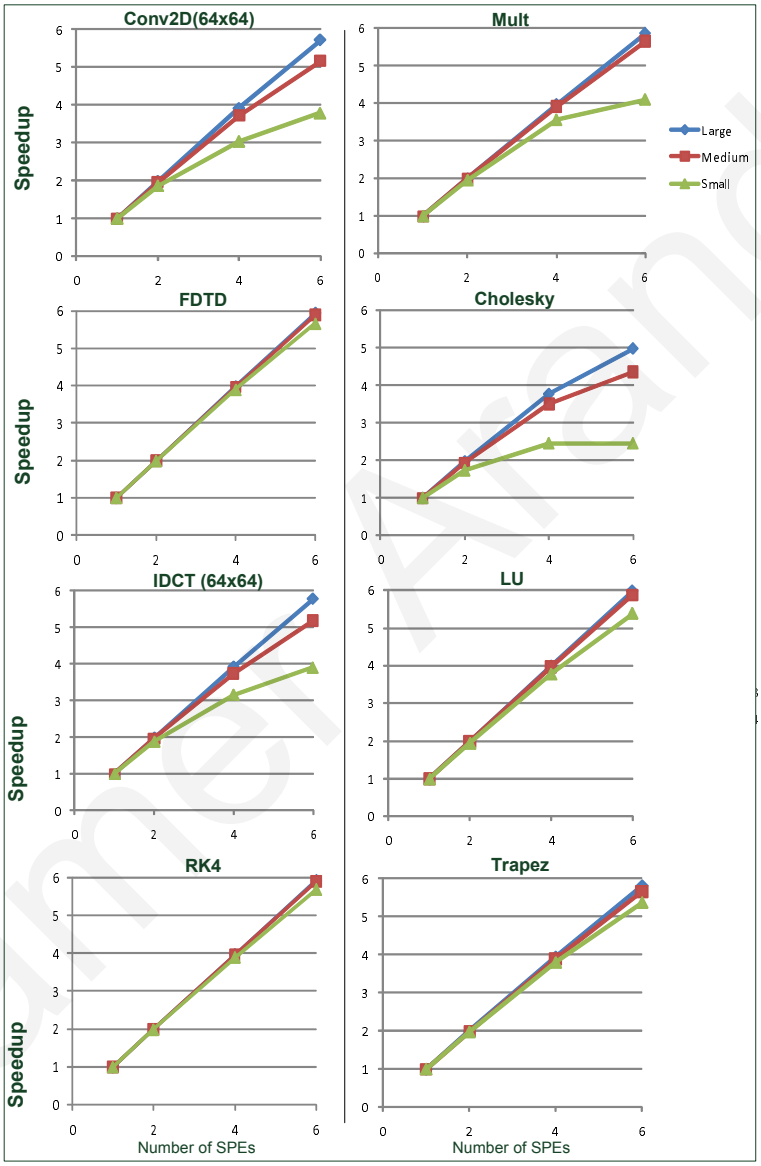


Figure 79: Effect of problem sizes on performance

7.2.4 Problem Size

Figure 79 depicts the results of executing 8 of the benchmarks for the three problem sizes. The results show that the system generally scales well across the range of the benchmarks achieving almost linear speedup for the large problem sizes, as large problem sizes result in longer execution time, which amortizes initialization and parallelization overheads. We expect DDM-VM$_c$ to scale well in real life (scientific) applications since our benchmarks are representative of such applications and the typical real-world sizes employed for such application are bigger that our *Large* problem size.

7.2.4.1 Overall Performance and Comparison

In this section, we report the GFLOPs performance results of three computationally intensive applications, MatMult, Cholesky and Conv2D and compare them with the StarSs implementation targeting the Cell processor: CellSs [18, 94] and Sequoia [41] platforms that target the Cell processor.

The result for CellSs were obtained by executing the MatMult and Cholesky applications found in the latest release of CellSs platform V2.2 on a PS3. The two applications use the same computational kernels we have used for our applications. For these results we have used the following combination of parameters which produced the best performance. For the MatMult application (ExFQ=3, Throttling Limit=8, Locality Enabled, Cache Size=128KB and using the *direct* SM technique). For the Cholesky and Conv2D applications (ExFQ=3, Throttling Limit=3, Locality Enabled, Cache Size=128KB and using the *direct* SM technique).

Figure 80 depicts the GFLOPs performance results for the MatMult and Cholesky applications and compares the performance with CellSs. The results show that for the MatMult application DDM-VM$_c$ performs very well achieving an average of 88% of the theoretical peak performance

for the 2048 size and an average of 86% and 76% for the 1024 and 512 sizes respectively, scaling almost linearly on the first two sizes. The results for Cholesky are not as good as MatMult for the smaller sizes due to the complex nature of the application, however when the size becomes 2048 the application scales very well achieving a speedup of 5 on 6 SPEs.

The comparison results in Figure 80 demonstrate that DDM-VM$_c$ outperforms CellSs for the entire range for both applications. DDM-VM$_c$ achieves an average improvement of 80% for the 512 size, 28% for 1024 and 19% for the 2048 size for MatMult. An improvement of 213% for 512, 99% for 1024 and 23% for 2048 is achieved for Cholesky. We attribute this to the fact that although CellSs schedules annotated tasks at run-time based on data-dependencies like our model, in contrast with ours which creates the dependency graph statically, CellSs builds it at run-time. This can contribute more delay to the critical path of the application than in our model. Moreover, CellSs makes only part of the graph available to the scheduler and consequently a fraction of the concurrency opportunities in the applications is visible at any time. DDM-VM$_c$ achieves the best improvement v.s. CellSs for the smaller problem sizes, which indicates that it introduces less overhead for exploiting concurrency.

Figure 81 compares the performance of DDM-VM$_c$ and Sequoia for the MatMult and Conv2D applications. The results for Sequoia were obtained by executing the MatMult and Conv2D applications found in the latest release of Sequoia that targets the Cell (V0.9.5) on a PS3. To preserve fairness we have used the same computational kernels used in the Sequoia applications for our applications as well. The results show that DDM-VM$_c$ achieves an average of 25% and 12% performance improvement for Conv2D and MatMult, respectively.

We find these results as an indication of the efficiency of the DDM-VM$_c$ and its ability to perform favorably with other platforms on the Cell.

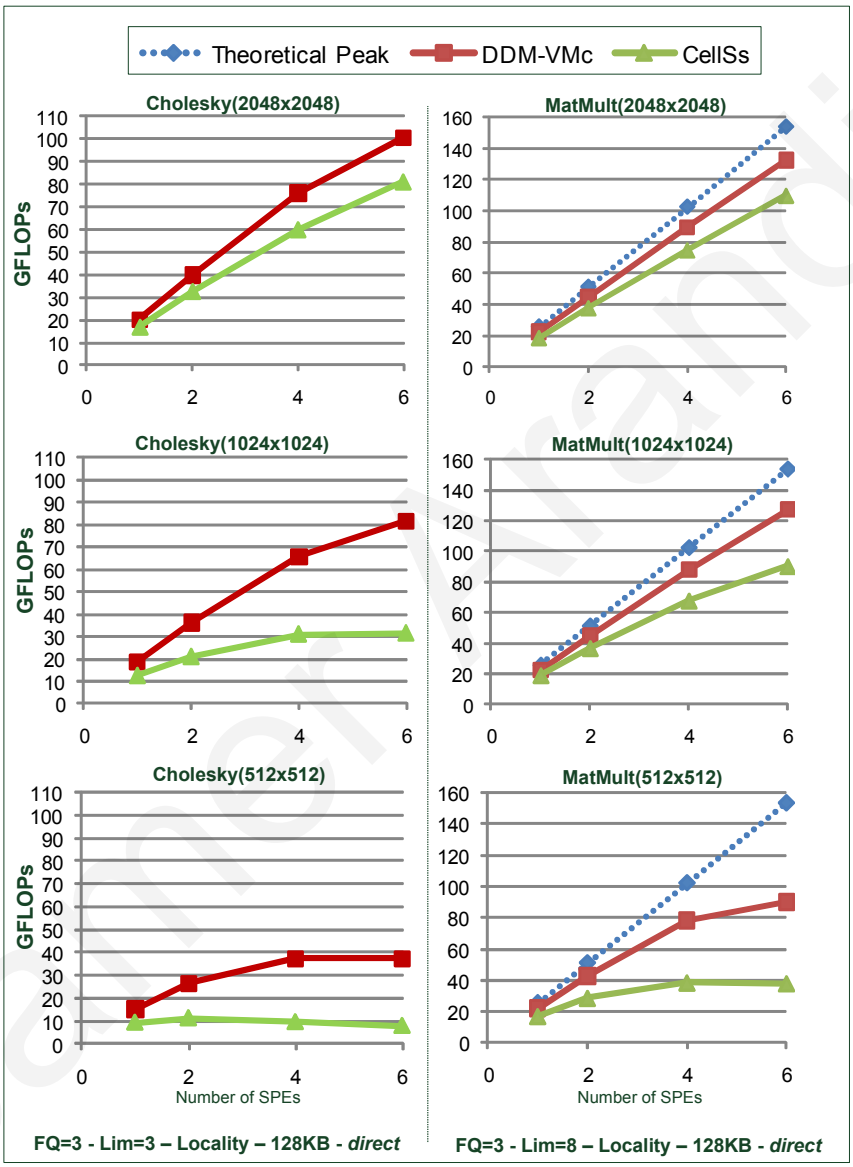


Figure 80: Comparison of DDM-VM$_c$ and CellSs Performance for the MatMult and Cholesky applications

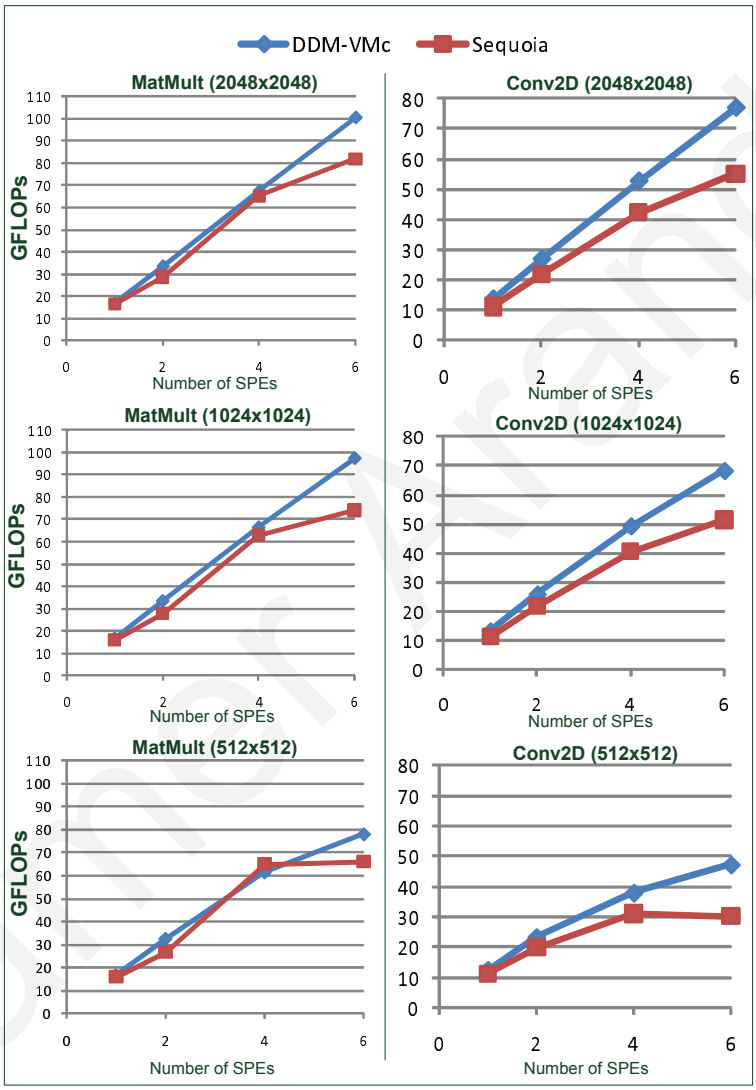


Figure 81: Comparison of DDM-VM$_c$ and Sequoia Performance for the MatMult and Conv2D applications

7.2.5 Distributed DDM-VM$_c$ Execution

For the evaluation of distributed DDM-VM$_c$ execution we used a cluster of four PS3 nodes. The benchmarks we executed contains applications that don't communicate during the execution (Conv2D, IDCT and MatMult), ones that communicate few values (Trapez) and ones with heavy inter-node communication (LU and Cholesky). For all the benchmarks working on matrices we have used blocks of 64x64 except for the Conv2D benchmark in which we used 96x96 blocks. For the Cholesky benchmark we used scalar computational kernels instead of the vectorized ones as the latter proved too fine-grained for the application to scale. We denote the version using the scalar kernels as Cholesky-S. In our experiments we have utilized 1, 2, 4 and 6 SPEs per node, which resulted in 4, 8, 16 and 24 total SPEs in the system, respectively. Moreover, we have used two input sizes per benchmark. Figure 82 illustrates the speedup results.

The results show that for the largest input size the system achieves an average of 80% of the maximum possible speedup for all the benchmarks, which is a very good result. Analyzing the results further, it is clear that as the input size increases the system scales better: the average speedup (on all the benchmarks) utilizing all the SPEs is 13.4 out of 24 for the smaller input size and 16.54 out of 24 for the larger input size. This is expected as larger problem sizes allow for amortizing the overheads of the parallelization. The limited main memory available on the PS3 nodes (256MB) precluded us from using larger input sizes. However, this limitation does not exist on other commercial products powered by the Cell processor, thus allowing the DDM-VM$_c$ to scale further on such systems.

Note that compared to single-node execution larger input sizes (on all the benchmarks) and larger granularities (on Conv2D and Trapez) are needed for the system to scale due to the additional latencies introduced by the network data and synchronization messages transfer.

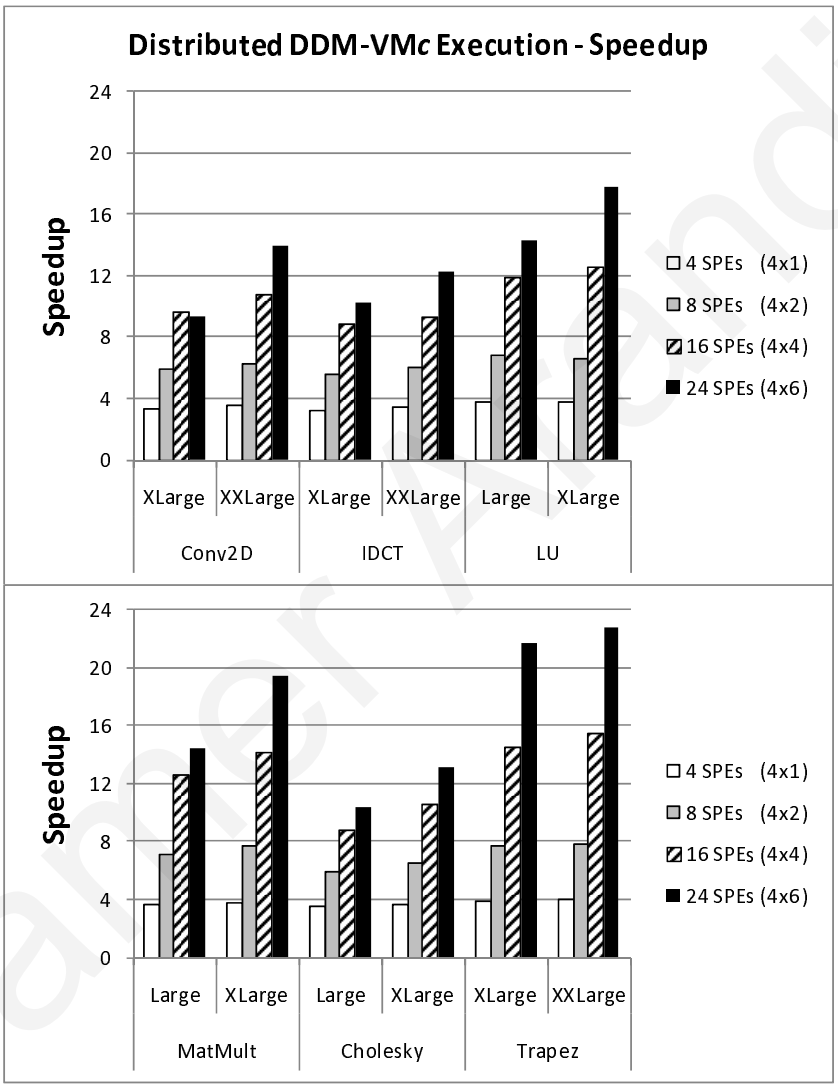


Figure 82: Distributed DDM-VM$_c$ Execution - Speedup

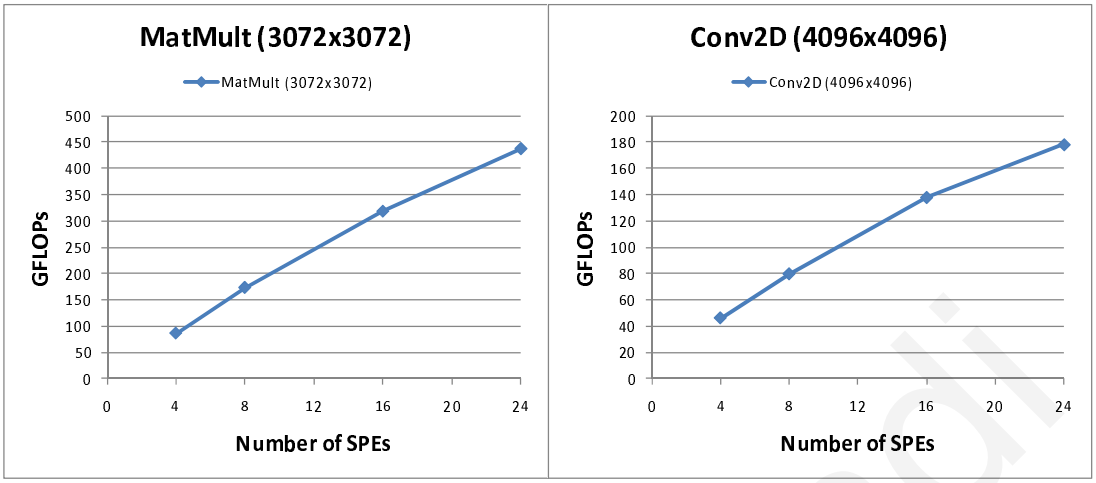


Figure 83: GFLOPs performance results for MatMult and Conv2D

Figure 83 reports the GFLOPs performance results for the two computationally intensive benchmarks MatMult and Conv2D.

The results illustrate that utilizing all the SPEs on the four nodes the system delivers an impressive 0.44 TFLOPs for the MatMult benchmark and 178 GFLOPs for the Conv2D benchmark, which demonstrates the efficiency of the distributed execution on the DDM-VM$_c$.

7.3 The DDM-VM$_s$ Evaluation

In this section we present the evaluation of the DDM-VM$_s$. We provide a comprehensive performance evaluation using the benchmarks and examine the effect of thread granularity and input size on performance. We also evaluate the performance of the runtime-determined dependency resolution approach we propose. Finally, we present the evaluation of the distributed DDM-VM$_s$ execution.

Experimental Setup

The DDM-VM$_s$ implementation runs on a 12-core machine composed of two Six-Core AMD Opteron Processors with 64KB L1 D-Cache, 64KB L1 I-Cache, 1MB unfied L2 cache and 6MB unified L3 cache and a 32 GB of RAM. The cores run at 800 MHz with the Ubuntu Linux 2.6.31 server edition as the OS.

We used the same benchmark suite used in the evaluation of the distributed DDM-VM$_c$. The characteristics of the benchmarks are presented in Table 4. Note that although the applications are the same their characteristics (mainly the granularity) differ due to the vectorization of the computational kernels in the case of the DDM-VM$_c$ and the fact that the code is compiled with different compilers. Moreover, the systems used for evaluating the DDM-VM$_s$ has more main memory compared to that on the PS3, which enables us to use larger input sizes.

All of the benchmarks were coded in C using the DDM-VM *macros* and compiled using the GCC 4.4.3 compiler. All reported speedup results are relative to the execution time of the best corresponding (non-DDM) sequential code on one core. Note that the maximum possible speedup is 11, since we reserve one core out of the 12 cores for the execution of the TSU.

Table 4: The benchmarks suite characteristics - DDM-VM$_s$

Benchmark	Decription	Average Granularity of Benchmark Threads			Problem Size				
		Granularity	Execution Time		Small	Medium	Large	XLarge	XXLarge
			System-2	System-1					
MatMult	Blocked Matrix Multiplication	8x8	1 μs	1 μs	128x128	512x512	2048x2048	4096x4096	8192x8192
		16x16	8 μs	11 μs					
		32x32	53 μs	72 μs					
		64x64	387 μs	528 μs					
		128x128	3333 μs	4540 μs					
MatMult	Blocked Matrix Multiplication - Coarge-grained	8x8	110 μs	140 μs	128x128	512x512	2048x2048	4096x4096	8192x8192
		16x16	535 μs	681 μs					
		32x32	1920	2309					
		64x64	7250 μs	8436 μs					
		128x128	28500 μs	36350 μs					
Cholesky	Blocked Cholesky Factorization	8x8	1 μs	1 μs	128x128	512x512	2048x2048	4096x4096	8192x8192
		16x16	3 μs	4 μs					
		32x32	16	22 micros					
		64x64	134 μs	182 μs					
		128x128	916 μs	1240 μs					
LU	Blocked LU Decomposition	8x8	1 μs	1 μs	128x128	512x512	2048x2048	4096x4096	8192x8192
		16x16	8 μs	11 μs					
		32x32	52 μs	73 μs					
		64x64	380 μs	520 μs					
		128x128	2918 μs	3975 μs					
Conv2D	9x9 convolution filter	8x8	10 μs	14 μs	128x128	512x512	2048x2048	4096x4096	8192x8192
		16x16	39 μs	54 μs					
		32x32	157 μs	214 μs					
		64x64	626 μs	855 μs					
		128x128	2500 μs	3416 μs					
IDCT	Inverse Discrete Cosine Transform	8x8	less than 1μs	less than 1μs	128x128	512x512	2048x2048	8192x8192	16384x18384
		16x16	1 μs	1 μs					
		32x32	3 micros	4					
		64x64	12 μs	17 μs					
		128x128	49 μs	68 μs					
Trapez	Trapezoidal Rule of Integration	variable	variable	variable	168M steps	337M steps	675M steps	1350M steps	–

7.3.1 Thread Granularity

To assess the effect of thread granularity on performance we executed the benchmarks with varying thread granularities for the *large* input size. Table 4 reports this information for every benchmark. The speedup results are depicted in Figure 84. The baseline for the speedup is the best sequential (non-DDM) execution among all the granularities.

The results demonstrate that the performance improves as the granularity increases, as higher granularities amortize better the scheduling overheads of the TSU. This result confirms the findings of 7.2.3.1.

7.3.2 Input Size

Figure 85 depicts the speedup results of executing the benchmarks for the *small*, *medium* and *large* input sizes. For all the benchmarks working on matrices the 64x64 granularity is used. The results demonstrate that as input sizes increases the performance improves. This is expected, as large problem sizes result in longer execution time, which amortizes initialization and parallelization overheads. This result confirms the findings of 7.2.4.

7.3.3 Overall Performance

Figure 86 depicts the speedup results of executing the *large* input size for the 64x64 granularity for all the benchmarks.

The results of executing all the benchmarks demonstrate that overall, the system scales well over the range of the benchmarks and achieves - when utilizing all the cores - an average speedup of 9.6 out of 11 (the maximum possible speedup), which indicates the efficiency and scalability of the system.

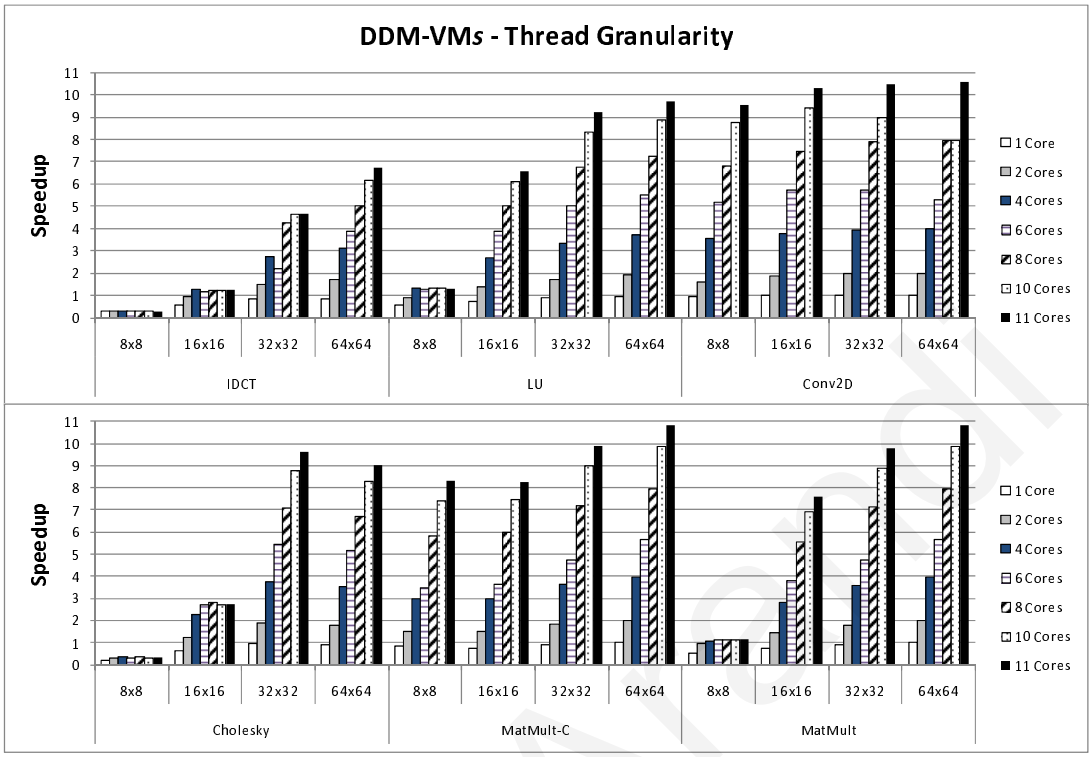


Figure 84: Effect of thread granularity on performance

7.3.4 Runtime Dependency Resolution

In this section we evaluate the approach we utilize for handling runtime-determined dependencies. We study the effect of the overheads of the I-Structure operations on the performance by comparing 3 versions of the MatMult, Cholesky and LU benchmarks:

- The first version utilizes the compile-time approach for resolving the dependencies in the program.
- The second version combines both approaches; part of the dependencies are resolved at compile-time and the rest are resolved at runtime (using *I-Fetch* and *I-Store* operations).
- The third version utilizes the runtime approach for resolving the dependencies.

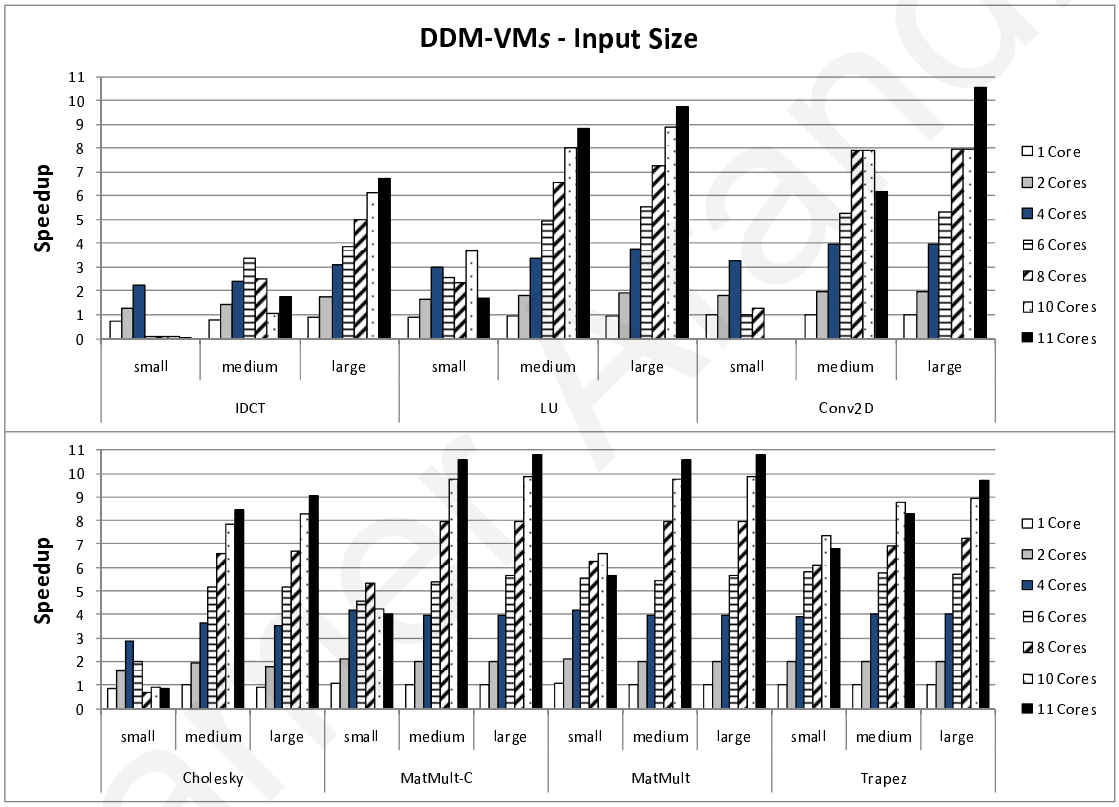


Figure 85: Effect of problem sizes on performance

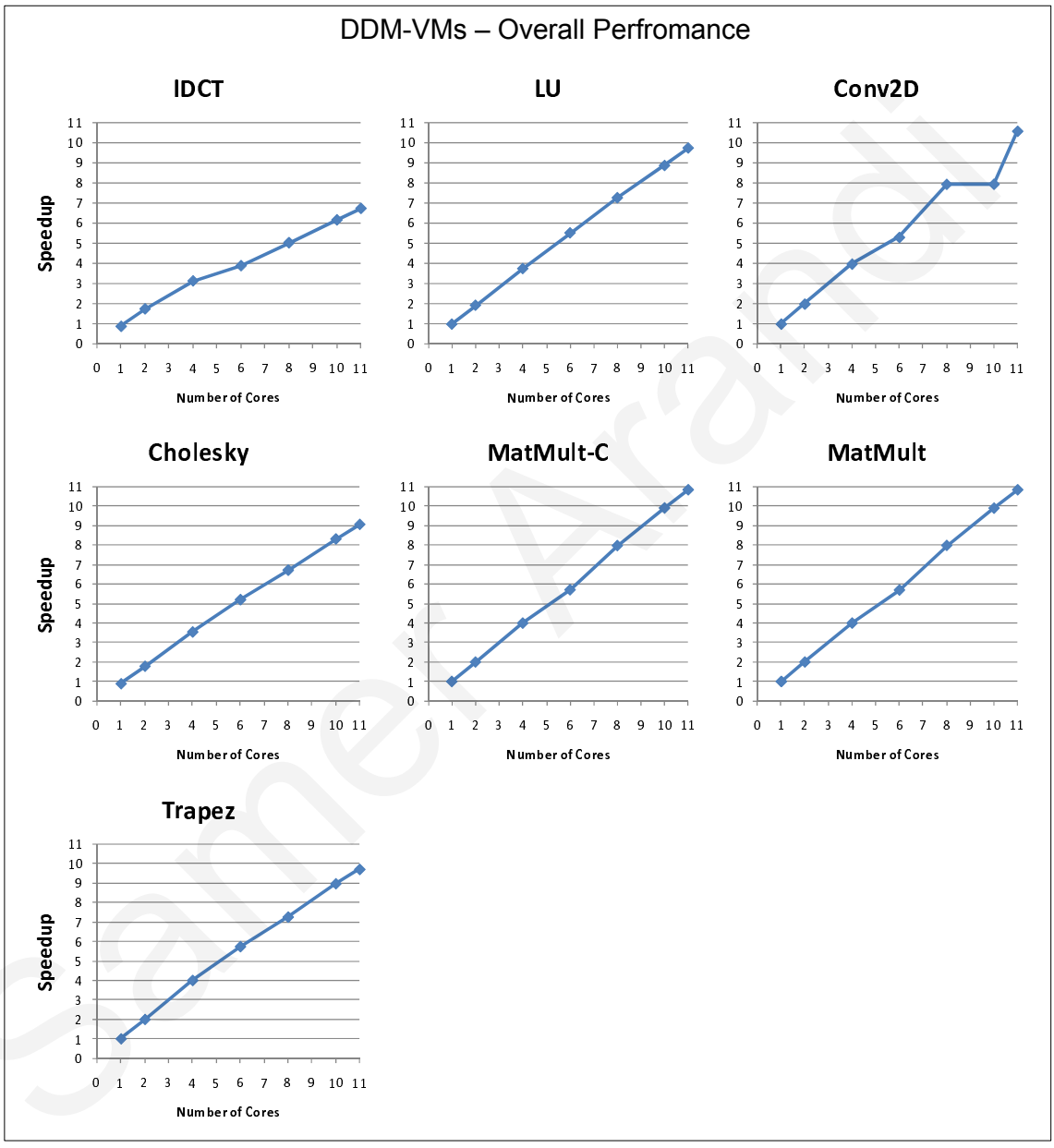


Figure 86: DDM-VM$_s$ overall performance

We compare the performance of the 3 versions for various thread granularities (16x16, 32x32 and 64x64). The results are depicted in Figure 87. We refer to the 3 versions in the figures and the subsequent text as *C-D*, *CR-D* and *R-D*, respectively. Note that for the MatMult benchmark, only the first and third versions are available, as the threads in this program have one data dependency.

The results demonstrate that, as expected, the best performance is delivered by the version utilizing the compile-time approach (C-D), followed by the version utilizing the combination of the compile-time and runtime approaches (RC-D).

The results show that the performance loss (relative to the compile-time version) is higher for lower granularities and decreases as we increase the granularity. For example, when using 10 cores in the Cholesky application, the performance loss when utilizing the runtime approach for all the dependencies (R-D) is 43% for the smallest granularity (16x16) compared to 13.6% for the largest granularity (64x64). When utilizing a combination of the two approaches (CR-D) the loss is 14.8% for the smallest granularity compared to 2.2% for the largest granularity. The same observation applies to the two other benchmarks. This is clearly demonstrated in Figure 88, which depicts the execution time of the R-D and RC-D versions normalized to the execution time of the C-D version.

The reason is that as we increase the granularity of the threads by increasing the size of the blocks the threads operate on, the total number of blocks decreases and so does the total number of thread invocations. Consequently the number of *I-Fetch* and *I-Store* operations decreases, thus reducing the overheads. Moreover, increasing the granularity of the threads amortizes the I-Structure operations overheads.

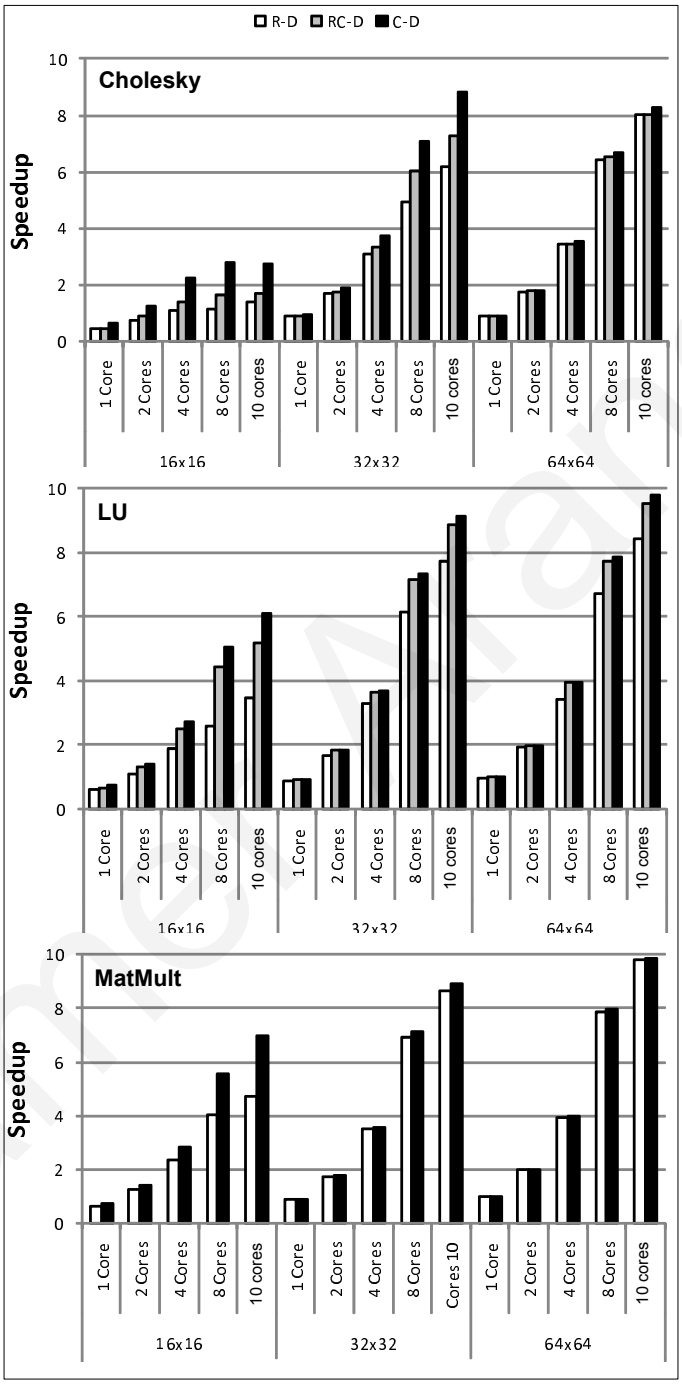


Figure 87: Speedup comparison: runtime-determined dependencies (R-D) v.s. runtime & compile-time determined dependencies (RC-D) v.s. compile-time determined dependencies (C-D) approaches

The results demonstrate that utilizing the proposed technique (for part or all of the data dependencies in the evaluated programs) achieves acceptable performance compared to the compile-time approach, whilst utilizing thread granularities in the range we normally utilize in DDM-VM programs.

7.3.5 Distributed DDM-VM$_s$ Execution

For the evaluation of distributed DDM-VM$_s$ execution we used two cluster: The first is composed of two of the 12-core AMD machines described previously. We refer to this cluster as *System-1*. The second is composed of four quad-core AMD machines running at 3000 MHz, which have 64KB L1 D-Cache, 64KB L1 I-Cache, 2MB unified L2 cache and 6MB unified L3 cache and 4GB of RAM. We refer to this cluster as *System-2*. Both clusters are connected using an off-the-shelf Giga-bit Ethernet switch with a latency of approximately 200 μs.

We used the same benchmarks used for evaluating the distributed DDM-VM$_c$ execution in 7.2.5. For all the benchmarks working on matrices we have used blocks of 128x128. In our experiments we utilized 1, 4, 8 and 11 cores per node for System-1 cluster, which resulted in 2, 8, 16 and 22 total cores in the system, respectively. For the System-2 cluster we utilized 1, 2 and 3 cores per node, which resulted in 4, 8 and 12 total cores, respectively (remember that we always reserve one core for the TSU execution and thus the maximum number of utilized cores is 11 on the 12-core machine and 3 on the 4-core machine). We have used two input sizes per benchmark. Figures 89 & 90 illustrate the speedup results for both clusters.

The results show that for the largest input size the system achieves an average of 80% and 84% of the maximum possible speedup for the System-1 and System-2 clusters, respectively, which is a very good result.

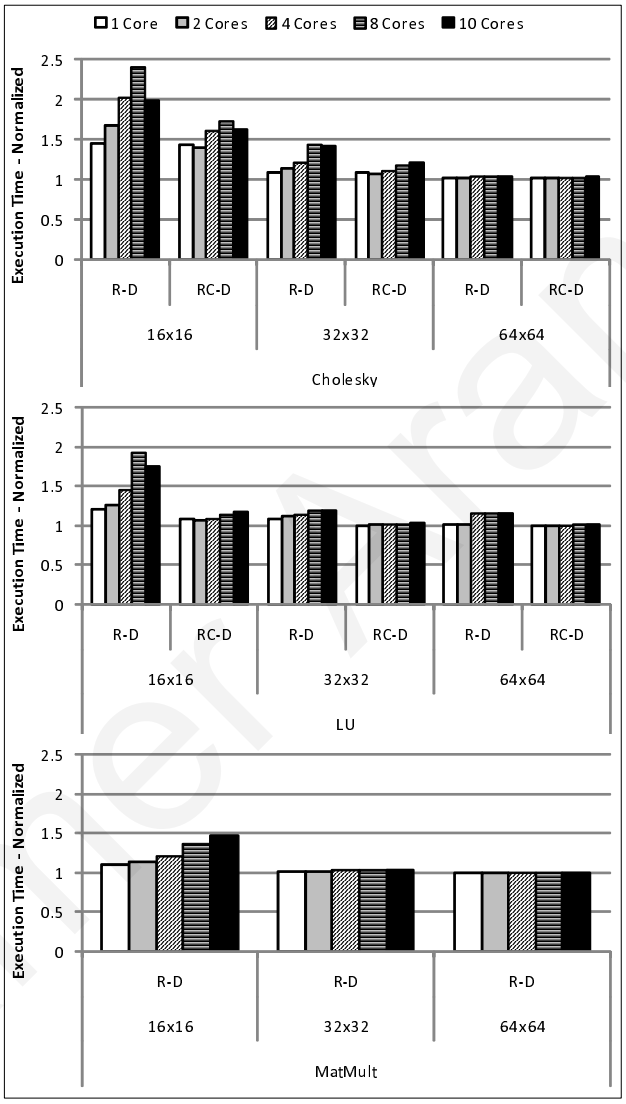


Figure 88: Execution time comparison: execution time using the runtime-determined dependencies approach v.s. the runtime-determined & compile-time determined dependencies approach normalized to the execution time using the compile-time determined dependencies approach

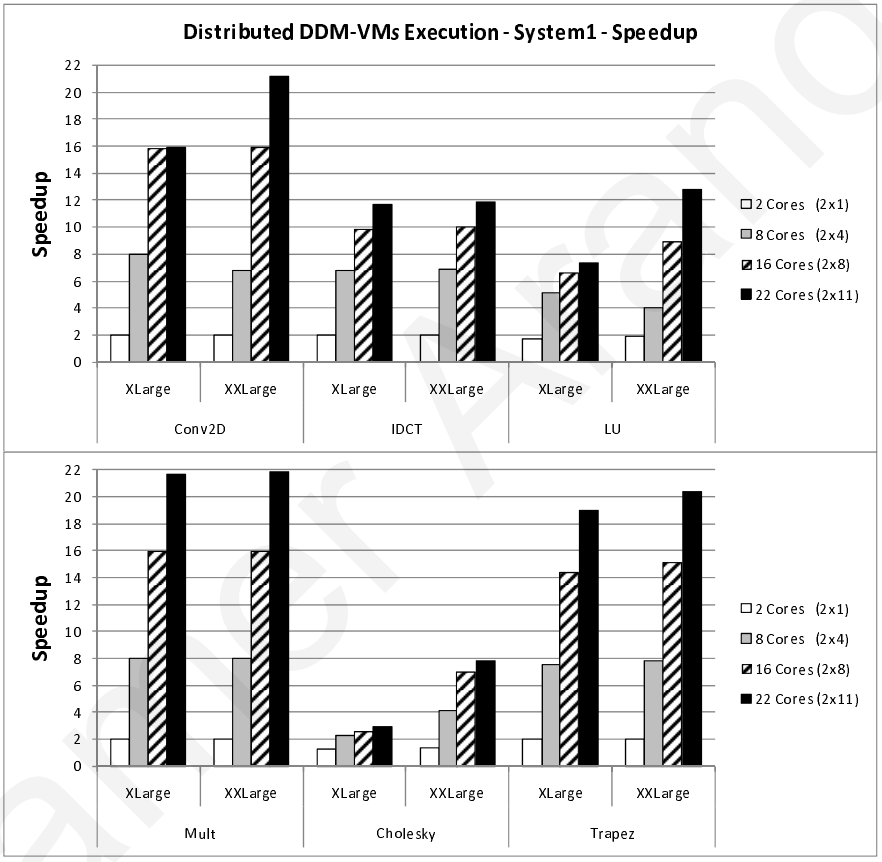


Figure 89: Distributed DDM-VM$_s$ Execution (System-1) - Speedup

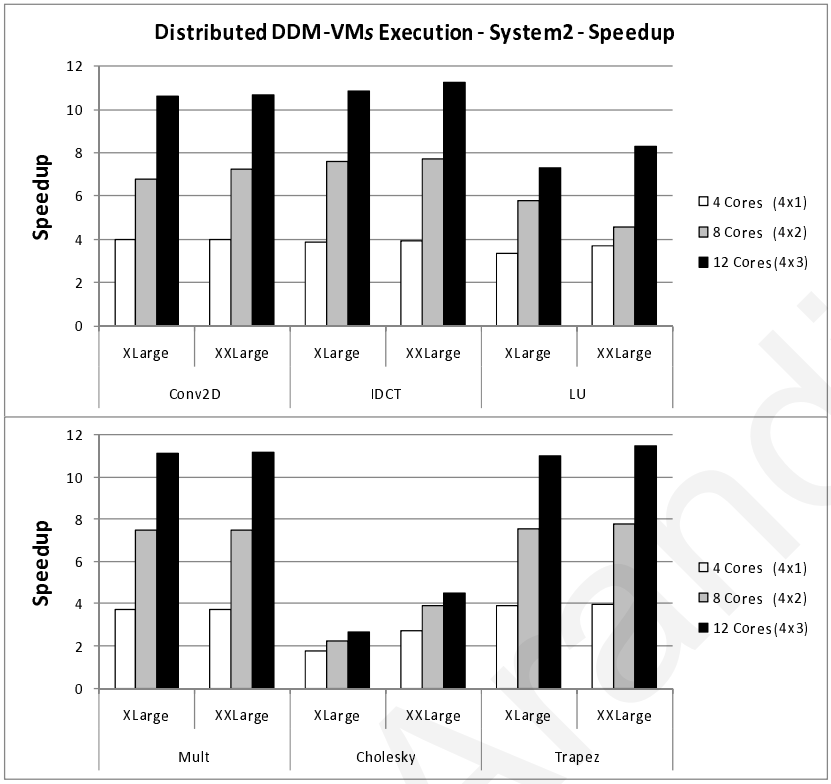


Figure 90: Distributed DDM-VM$_s$ Execution (System-2) - Speedup

Table 5: Distributed DDM-VM$_s$ Execution Results - Summary

	System-1 Cluster		System-2 Cluster	
	Smaller Input Size	Larger Input Size	Smaller Input Size	Larger Input Size
Average Speedup Percentage	74%	80%	79%	84%
Average Speedup (utilizing all cores)	13.1/22	16/22	8.9/12	9.5/12

Similar to the results of the distributed DDM-VM$_c$ execution in 7.2.5, the system scales better as the input size increases as this allows for amortizing the overheads of the parallelization. The average speedup utilizing all the cores is 13.1 out of 22 for the smaller input size and 16 out of 22 for the larger input size for the System-1 cluster. The average speedup on the System-2 cluster is 8.9 out of 12 for the smaller input size and 9.5 out of 12 for the larger input size. The results are summarized in Table 5.

As noted in 7.2.5 larger input sizes and granularities (compared to single-node execution) are needed for the system to scale due to the additional latencies introduced by the network data and synchronization messages transfer.

The Cholesky benchmark yields the least performance as its threads exchange data heavily across the nodes and so is affected to a great extent by the large latency of the the network. The LU benchmark similarly has a heavy inter-node data exchange, however, because its threads have a larger granularity compared to Cholesky's (for the same block size), the TSU has a better chance of overlapping the network latencies, thus yielding better performance.

Chapter 8

Future Work and Conclusion

8.1 Future Work

The first focus of our future work is the further improvement of the the DDM-VM programming toolchain. In addition to pursuing the efforts on the GCC auto-parallelizing compiler (described in Section 5.8), we plan to develop a source-to-source compilation tool that facilitate using the Concurrent Collections (CnC), a declarative parallel programming language, to program the DDM-VM.

The second focus on the future work is the further improvement of the DDM-VM performance. On this front we plan to support dynamic scheduling in distributed execution to improve the performance of applications benefiting from load-balancing. Moreover, we target supporting prefetching on the DDM-VM$_s$ targeting architectures with hardware-managed memories that support prefetching. We describe our future work efforts in detail in the following sections.

8.1.1 Concurrent Collections Source-to-Source Compiler

Concurrent Collections [22, 23] is a declarative parallel programming language, with similar semantics to DDM. It allows programmers who lack experience in parallelism to express their

parallel programs as a collection of high-level computations called *steps* that communicate via single-assignment data structures called *items*. *Steps* and *items* are uniquely identified by *tags*. The major CnC constructs match the DDM constructs: the CnC *steps* correspond to the DDM threads, as both represent the unit of execution and apply single-assignment across *steps*/threads while allowing side-effects locally within a *step*/thread. The control and data dependence relationships amongst the steps, manifested in the *items* and *tags* that are produced and consumed, correspond to the producer-consumer relationships (the *meta-data*) of the DDM threads.

This correspondence facilitates translating CnC programs into DDM-VM programs. This allows programmers to write their applications in CnC and efficiently handles the low-level details of the parallel execution including the memory management on architectures with software-managed memory hierarchy.

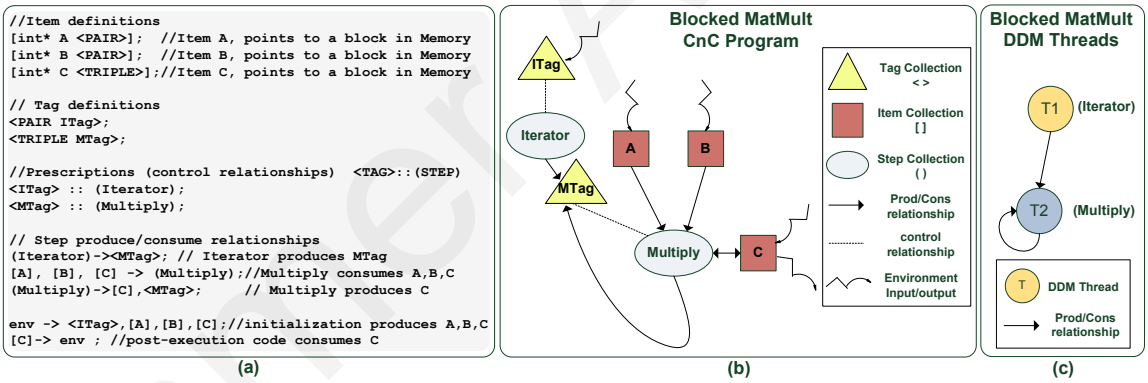


Figure 91: The blocked Matrix Multiplication application. (a) Textual representation of the CnC program (b) Graphical representation of the CnC program. (c) Equivalent DDM dependency graph.

To this end, a CnC source-to-source compiler is being developed, which parses the CnC program and generates the corresponding DDM threads code and augments it with calls to the DDM-VM runtime.

Figures 91-a and 91-b illustrate the textual and graphical representations of a CnC program implementing the Blocked Matrix Multiplication. The program consists of two *steps* accessing

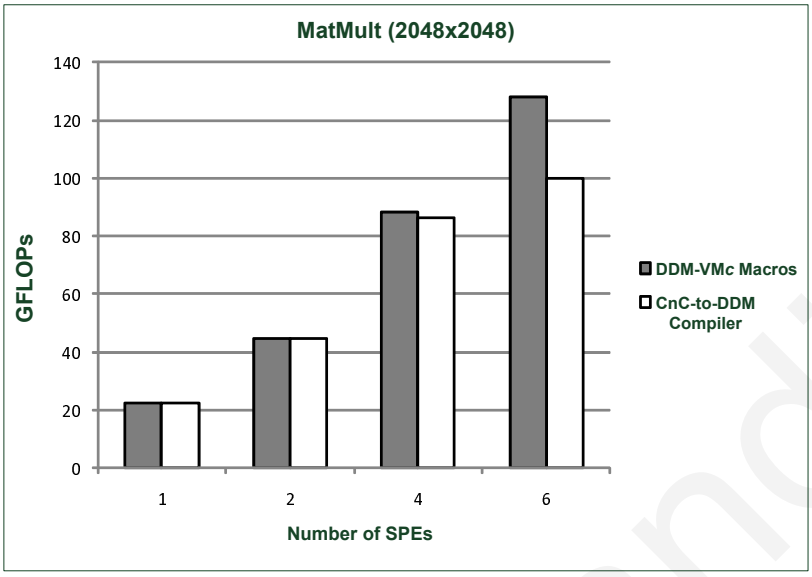


Figure 92: Performance comparison between the *macro*-coded and compiler-generated versions of the matrix multiplication program

three *items*, in addition to two *tags*. Figure 91-c depicts the dependency graph of the equivalent DDM program where each *step* was mapped into a DDM thread. The Figure also depicts the dependencies between the threads. Next, we presents the preliminary evaluation results for the CnC compiler.

Preliminary Results

We compare the performance of two versions of the Matrix Multiplication, one coded using the DDM-VM *macros* vs. one generated using the preliminary version of the CnC compiler we are developing. Both versions are run on the DDM-VM$_c$ and the results are depicted in Figure 92.

The results show that the compiler-generated version is performing on par with the macro-coded one achieving an impressive 86.5 GFLOPS for 4 SPEs. When the number of SPEs is six the performance of the compiler-generated version drops. We attribute this to an inefficient implementation of the *hashmap* structure we use to represent CnC data *items* in the generated program. A

more efficient implementation will be developed. Nevertheless, we find these preliminary results very encouraging to pursue work in this direction.

8.1.2 Supporting Dynamic Scheduling in Distributed Execution

The work presented in this thesis explored a *static* scheme for distributing (or mapping) threads to cores, in which the mapping is determined at compile time and does not change during the execution. For the relatively small granularities of the utilized benchmarks threads, this scheme delivers the best performance. In this section we discuss the planned support for *dynamic* distribution, which can utilized for improving the performance of applications with much larger thread granularities that benefit from load-balancing.

8.1.2.1 Thread Scheduling Unit (TSU)

TSU Structures

To support the dynamic mapping scheme the TSU allocates the following three structures:

1. **Scheduling Table (ST)**: This structure keeps track of where each invocation of a thread has been scheduled (on which node/core).

2. **Load Table (LT)**: This structure has an entry per node. It records the number of threads currently scheduled to run on that node, in addition to any other information that can be used when taking the scheduling decision (for example power and thermal information).

3. **Dynamic Data Directory (DDD)**: This structure keeps track of the addresses of the input data associated with the thread invocations scheduled to execute on the current node.

TSU Operations

Whenever a decrement RC request in the AQ is processed, the TSU interrogates the ST to get the identifier of the core where this invocation is scheduled. If the core is on the local node the request is processed normally. If the core belongs to a remote node the request is forwarded to that node. If no information on that invocation is available in the ST, the TSU requests information from the other TSUs in the system. Once the information is received, it updates the ST and forwards the request accordingly if needed. If all the TSUs have no information (this is the first decrement RC request for that invocation) a scheduling decision is taken according to the scheduling criterion (e.g. load-balancing). The decrement request is then forwarded accordingly and the ST is updated and the rest of the TSUs are informed to update their LTs and possibly their STs. Once a thread invocation finishes execution the rest of the nodes are informed to update their tables. The NIU supports the exchanging of messages that is needed for implementing all the previously described interactions amongst the nodes.

8.1.2.2 Program Data

The thread input/output data is allocated dynamically at runtime, because it is only then that the core where each thread invocation is mapped, is specified. Consequently, instead of specifying the GAS address, the Data Frame Pointer (DFP) describing the output data specifies: (i) the consumer thread identifier and its *context* and (ii) an index specifying which consumer input this data maps to (the first input, second input, etc). Similarly, instead of specifying the GAS address, the DFP describing the input data merely specifies the index of the input data.

The only exception to the above are the threads that consume initialization data or ones that write produced data at the end of the program into a previously allocated result buffer, as in these cases the GAS address of the data is known *apriori* and so is directly specified in the DFP.

When a producer thread finishes execution, and the data is *forwarded* to a remote consumer with the decrement RC request, the receiving node uses the message header field describing the size of the forwarded data to dynamically allocates a buffer to hold the data in its memory. The TSU associates the address of the data, with the consumer thread identifier & *context* and the data input index in the Dynamic Data Directory table. When the RC of a consumer reaches zero, the table is used to retrieve the addresses of this invocation data and the rest of the TSU activities proceed as described previously. Once the thread finishes execution its associated table entries are deallocated to minimize the size of the table.

8.1.3 Supporting Prefetching on the DDM-VM$_s$

The current design of the S-CacheFlow module in the TSU supports prefetching for architectures with software-managed memory hierarchies in the DDM-VM$_c$ implementation. To improve the performance on architectures with hardware-managed caches, we target supporting prefetching on the DDM-VM$_s$ by applying the same CacheFlow principles. In such architectures the S-CacheFlow allocation and eviction tasks are handled by the memory controller hardware, however, we have control over the prefetching of data into the cache. We discuss the support for prefetching on x86 64-bit architectures without loss of generality.

The x86 64-bit family of processors supports a number of prefetching instructions [3, 1, 4]. Although the general behavior of such instructions is defined by the architecture, the processor implementations can ignore or change how these instructions operate [37].

The most important factors to our implementation is the cache level affected by prefetching and the configuration of that level (e.g. unified v.s. private or inclusive v.s. exclusive). One important factor that must be taken in consideration is the effect of the hardware prefetchers, which could interfere with the S-CacheFlow prefetching.

One fundamental aspect of the implementation is where to issue the prefetch instruction. This can be issued by the TSU or by the runtime threads on the cores where the execution of the threads is taking place. We anticipate the former would work only if the prefetch instruction loads the data into a unified cache level (typically L2 and higher) and the latter would work in either case (unified or private). For that reason our initial design opts for the latter. To issue the prefetches the runtime uses the information of the thread input/output data in the ExFQ.

8.2 Conclusion

In this thesis we proposed adopting the Data-flow model as the basis for an execution model that exploits the resources of multi-core architectures. We designed and implemented the DDM-VM: a virtual machine that supports DDM execution on homogeneous and heterogeneous multi-cores for both single-node and distributed/multi-node systems. The DDM-VM utilizes Data-Flow concurrency for scheduling threads and efficient sequential execution within a thread, while optimizing the context management of the Dynamic Data-Flow tagging system.

In the context of this work we also proposed the use of a data-driven prefetching software cache for handling software-managed memory hierarchies. We presented the programming methodology and developed a number of alternative approaches facilitating the programming of the DDM-VM, in addition to a number of optimizations for improving the performance. We also proposed combining compile-time and run-time dependency resolution using helper/proxy threads and special I-Structures. This expands the class of programs that can be handled by the DDM-VM and has the potential to enhance the programmability and improve the yield of compilation techniques generating data-flow code.

The evaluation demonstrates that the two implementations of the DDM-VM (for both single-node and multi-node/cluster) scale well and tolerate latencies and synchronization overheads efficiently and achieve very good overall performance. When comparing the performance of the DDM-VM$_c$ implementation with two similar state-of-the-art systems (StarSs and Sequoia) using a number of computationally-intensive benchmarks, the DDM-VM$_c$ outperforms both systems and achieves 88% of the theoretical peak performance for one of the benchmarks. For the same benchmark the distributed DDM-VM$_c$ execution on a cluster of 4 machines achieves 0.44 TFLOPs.

The main contribution of this thesis is that we have implemented Dynamic Data-Flow principles efficiently on off-the-shelf multi-core systems as a virtual machine that outperforms similar systems and delivers high-performance. This result strengthens the case that hybrid models that combine Data-Flow concurrency with efficient control-flow execution are candidates for adoption as the basis of a new execution model for Multi-core systems.

We conclude this thesis by summarizing the contributions:

- The development of the Data-Driven Multithreading Virtual Machine (DDM-VM), an efficient virtual machine that supports Data-Driven Multithreading execution on multi-core systems. The DDM-VM utilizes DDM scheduling for exploiting the resources of multi-core architectures and tolerating synchronization and memory latencies. The VM has two individually optimized implementations: The DDM-VM$_s$ tailored for homogeneous multi-cores and the DDM-VM$_c$ tailored for heterogeneous multi-cores. Both implementations utilize a unified programming representation and toolchain and implement a number of performance optimizations. The DDM-VM$_c$ is a high-performance implementation of DDM that achieves better performance than similar state-of-the-art systems. It is also the first heterogeneous implementation of DDM. It is developed for heterogeneous multi-core architecture with a host/accelerator organization and a software-managed memory hierarchy.

- The development of Software CacheFlow (S-CacheFlow), a fully-automated software prefetching cache with variable cache block sizes and explicit data locality optimizations for handling explicitly-managed memory hierarchies.

- The development of the support for distributed DDM execution across an off-chip network. The DDM-VM is the first DDM implementation supporting distributed DDM execution across a cluster of multi-core nodes.

- The development of the support for runtime-determined dependency resolution using specialized I-Structures. The DDM-VM is the first DDM implementation that supports parallel execution of code that contains consumer-producer dependencies that are only resolved at runtime. Compile-time and run-time dependency resolution can be utilized simultaneously, which combines the strengths of both approaches and expands the class of programs that can be mapped to the DDM model. It also has the potential to improve the programmability and enhance the yield of compilation methods generating data-flow code.

- The development of a number of performance optimizations and monitoring & visualization tools.

Bibliography

[1] *AMD64 Architecture Programmer's Manual Volume 1: Application Programming*, 3.09 edition, september 2003.

[2] May 2008. http://www.openmp.org.

[3] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, june 2011.

[4] *Software Optimization Guide for AMD Family 10h and 12h Processors*, 3.13 edition, february 2011.

[5] Maharaja (Raj) Pandian Eitan Peri Kurtis Ruby Francois Thomas Chris Almond Abraham Arevalo, Ricardo M. Matinata. *Programming the Cell Broadband Engine Architecture: Examples and Best Practices*. Number ISBN 0738485942. IBM Redbooks, 2008.

[6] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, 13(3):48–61, 1993.

[7] Jung Ho Ahn, William J. Dally, Brucek Khailany, Ujval J. Kapasi, and Abhishek Das. Evaluating the imagine stream architecture. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 14, Washington, DC, USA, 2004. IEEE Computer Society.

[8] Satoshi Amamiya, Masaaki Izumi, Takanori Matsuzaki, Ryuzo Hasegawa, and Makoto Amamiya. Fuce: the continuation-based multithreading processor. In *CF '07: Proceedings of the 4th international conference on Computing frontiers*, pages 213–224, New York, NY, USA, 2007. ACM.

[9] Samer Arandi and Paraskevas Evripidou. Programming multi-core architectures using dataflow techniques. In *SAMOS '10: Proceedings of the 10th International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, Samos, Greece, July 2010.

[10] Arvind and David E. Culler. Dataflow architectures. pages 225–253, 1986.

[11] Arvind and Kim P. Gostelow. The u-interpreter. *Computer*, 15(2):42–49, 1982.

[12] Arvind and Robert A. Iannucci. Two fundamental issues in multiprocessing. In *4th International DFVLR Seminar on Foundations of Engineering Sciences on Parallel Computing in Science and Engineering*, pages 61–88, New York, NY, USA, 1988. Springer-Verlag New York, Inc.

[13] Arvind and Vinod Kathail. A multiple processor data flow machine that supports generalized procedures. In *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture*, pages 291–302, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[14] Arvind and Rishiyur S. Nikhil. Executing a program on the mit tagged-token dataflow architecture. *IEEE Trans. Comput.*, 39(3):300–318, 1990.

[15] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11:598–632, October 1989.

[16] Eduard Ayguadé, Rosa M. Badia, Pieter Bellens, Daniel Cabrera, Alejandro Duran, Roger Ferrer, Marc González, Francisco D. Igual, Daniel Jiménez-González, and Jesús Labarta. Extending openmp to survive the heterogeneous multi-core era. *International Journal of Parallel Programming*, 38(5-6):440–459, 2010.

[17] Cedric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, Olivier Temam, A Group, and Inria Rocquencourt. Putting polyhedral loop transformations to work. In *In Workshop on Languages and Compilers for Parallel Computing (LCPC03), LNCS*, pages 209–225, 2003.

[18] Pieter Bellens, Josep M. Pérez, Rosa M. Badia, and Jesús Labarta. CellSs: a Programming Model for the Cell BE Architecture. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 86, New York, NY, USA, 2006. ACM.

[19] Pieter Bellens, Josep M. Perez, Felipe Cabarcas, Alex Ramirez, Rosa M. Badia, and Jesus Labarta. Cellss: Scheduling techniques to better exploit memory hierarchy. *Sci. Program.*, 17(1-2):77–95, 2009.

[20] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. Dag-consistent distributed shared memory. In *in Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA*, pages 297–308, 1996.

[21] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30:207–216, August 1995.

[22] Zoran Budimlic, Aparna M. Chandramowlishwaran, Kathleen Knobe, Geoff N. Lowney, Vivek Sarkar, and Leo Treggiari. Declarative aspects of memory management in the concurrent collections parallel programming model. In *DAMP '09: Proceedings of the 4th workshop on Declarative aspects of multicore programming*, pages 47–58, New York, NY, USA, 2009. ACM.

[23] Zoran Budimlic, Aparna M. Chandramowlishwaran, Kathleen Knobe, Geoff N. Lowney, Vivek Sarkar, and Leo Treggiari. Multi-core implementations of the concurrent collections programming model. In *CPC '09: 14th Workshop on Compilers for Parallel Computing*. Springer, 2009.

[24] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, William Yoder, and the TRIPS Team. Scaling to the end of silicon with edge architectures. *Computer*, 37(7):44–55, 2004.

[25] Draper J. M. Culler D. E. Yelick K. Brooks E. Carlson, W. W. and K. Warren. Introduction to upc and language specification. Technical report, University of California-Berkeley, 1999.

[26] Tong Chen, Haibo Lin, and Tao Zhang. Orchestrating data transfer for the cell/b.e. processor. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 289–298, New York, NY, USA, 2008. ACM.

[27] David E. Culler and Arvind. Resource requirements of dataflow programs. *SIGARCH Comput. Archit. News*, 16(2):141–150, 1988.

[28] David E. Culler, Seth Copen Goldstein, Klaus Erik Schauser, and Thorsten von Eicken. TAM - a compiler controlled threaded abstract machine. *J. Parallel Distrib. Comput.*, 18:347–370, July 1993.

[29] David E. Culler, Anurag Sah, Klaus E. Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: a compiler-controlled threaded abstract machine. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 164–175, New York, NY, USA, 1991. ACM.

[30] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. Tiny threads: A thread virtual machine for the cyclops64 cellular architecture. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 14 - Volume 15*, IPDPS '05, pages 265.2–, Washington, DC, USA, 2005. IEEE Computer Society.

[31] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[32] William J. Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J. Knight, and Ujval J. Kapasi. Merrimac: Supercomputing with streams. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, pages 35–, New York, NY, USA, 2003. ACM.

[33] Jack B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 362–376, London, UK, 1974. Springer-Verlag.

[34] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. *SIGARCH Comput. Archit. News*, 3(4):126–132, 1974.

[35] Keith Diefendorff. Compaq chooses smt for alpha. *Microprocessor Report*, 13(16), december 1999.

[36] Jack J. Dongarra, Rolf Hempel, Anthony J.G. Hey, and David W. Walker. A proposal for a user-level, message-passing interface in a distributed memory environment, 1993.

[37] Ulrich Drepper. What every programmer should know about memory, 2007.

[38] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–19, 1997.

[39] Alexandre E. Eichenberger, Kevin O'Brien, Kathryn M. O'Brien, Peng Wu, Tong Chen, Peter H. Oden, Daniel A. Prener, Janice C. Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, Michael Gschwind, Roch Archambault, Yaoqing Gao, and Roland Koo. Using advanced compiler technology to exploit the performance of the Cell Broadband EngineTM architecture. *IBM Syst. J.*, 45(1):59–84, 2006.

[40] Paraskevas Evripidou and Jean-Luc Gaudiot. A decoupled graph/computation data-driven architecture with variable-resolution actors. In *ICPP (1)*, pages 405–414, 1990.

[41] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the Memory Hierarchy. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83, New York, NY, USA, 2006. ACM.

[42] Nissim Francez. Distributed termination. *ACM Trans. Program. Lang. Syst.*, 2:42–55, January 1980.

[43] Jean-Luc Gaudiot. *On program decomposition and partitioning in data-flow systems*. PhD thesis, 1982. AAI8306041.

[44] Roberto Giorgi, Zdravko Popovic, and Nikola Puzovic. DTA-C: A decoupled multithreaded architecture for cmp systems. In *SBAC-PAD*, pages 263–270, 2007.

[45] Marc Gonzàlez, Nikola Vujic, Xavier Martorell, Eduard Ayguadé, Alexandre E. Eichenberger, Tong Chen, Zehra Sura, Tao Zhang, Kevin O'Brien, and Kathryn O'Brien. Hybrid access-specific software cache techniques for the cell be architecture. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 292–302, New York, NY, USA, 2008. ACM.

[46] Kim P. Gostelow and Robert E. Thomas. Performance of a simulated dataflow computer. *IEEE Trans. Comput.*, 29:905–919, October 1980.

[47] V. Gerald Grafe and Jamie E. Hoch. The epsilon-2 multiprocessor system. *J. Parallel Distrib. Comput.*, 10(4):309–318, 1990.

[48] Ed Grochowski, Ronny Ronen, John Shen, and Hong Wang. Best of both latency and throughput. In *ICCD '04: Proceedings of the IEEE International Conference on Computer Design*, pages 236–243, Washington, DC, USA, 2004. IEEE Computer Society.

[49] Gregory F. Grohoski. Machine organization of the ibm risc system/6000 processor. *IBM J. Res. Dev.*, 34:37–58, January 1990.

[50] Winfried Grunewald and Theo Ungerer. Towards extremely fast context switching in a block-multithreaded processor. *EUROMICRO Conference*, 0:0592, 1996.

[51] Babak Hamidzadeh, Lau Ying Kit, and David J. Lilja. Dynamic task scheduling using online optimization. *IEEE Trans. Parallel Distrib. Syst.*, 11:1151–1163, November 2000.

[52] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. In *Proceedings of the 22nd international symposium on Distributed Computing*, DISC '08, pages 350–364, Berlin, Heidelberg, 2008. Springer-Verlag.

[53] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[54] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, and et al. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *in Proceedings of the International Solid-State Circuits Conference*, pages 108–109, Feb 2010.

[55] ClearSpeed Inc. Clearspeed's csx processor architecture. *Whiter Paper; http://www.clearspeed.com/docs/resources*.

[56] RapidMind Inc. Cell be porting and tuning with rapidmind: A case study. *White Paper; see http://www.rapidmind.net/case-cell.php*.

[57] Intel. `http://www.intel.com/products/processor/core2duo/index.htm`.

[58] Intel. `http://www.intel.com/products/processor/core2quad/index.htm`.

[59] Intel. `http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_9485_13041,00.html`.

[60] Intel. `http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_13909,00.html`.

[61] Intel. Intel 80 cores by 2011. `http://techfreep.com/intel-80-cores-by-2011.htm`.

[62] Jack J. Dongarra Jakub Kurzak, Hatem Ltaief and Rosa M. Badia. Lapack working note 213: Scheduling linear algebra operations on multicore processors. Technical Report UT-CS-09-636, Computer Science Department, University of Tennessee, 2009.

[63] James A. Kahle, Michael N. Day, H. Peter Hofstee, Charles R. Johns, Theodore R. Maeurer, and David Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.

[64] Ron Kalla, Balaram Sinharoy, and Joel M. Tendler. Ibm power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

[65] Ujval Kapasi, William J. Dally, Scott Rixner, John D. Owens, and Brucek Khailany. The Imagine stream processor. In *Proceedings 2002 IEEE International Conference on Computer Design*, pages 282–288, September 2002.

[66] Krishna M. Kavi, Roberto Giorgi, and Joseph Arul. Scheduled dataflow: Execution paradigm, architecture, and performance evaluation. *IEEE Trans. Comput.*, 50(8):834–846, 2001.

[67] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[68] David A. Koufaty, Xiangfeng Chen, David K. Poulsen, and Josep Torrellas. Data forwarding in scalable shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 7:1250–1264, December 1996.

[69] Rakesh Kumar, Dean M. Tullsen, Norman P. Jouppi, and Parthasarathy Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, 2005.

[70] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[71] Costas Kyriacou. *Data Driven Multithreading using Conventional Control Flow Microprocessors*. PhD thesis, Dept. of Computer Science, University of Cyprus, 2005.

[72] Costas Kyriacou, Paraskevas Evripidou, and Pedro Trancoso. Cacheflow: A Short-Term Optimal Cache Management Policy for Data Driven Multithreading,. *Proc. EuroPar-04*, pages 561–570, Aug. 2004.

[73] Costas Kyriacou, Paraskevas Evripidou, and Pedro Trancoso. Data-Driven Multithreading Using Conventional Microprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 17(10):1176–1188, 2006.

[74] James Laudon, Anoop Gupta, and Mark Horowitz. Interleaving: a multithreading technique targeting multiprocessors and workstations. In *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 308–318, New York, NY, USA, 1994. ACM.

[75] Ben Lee and A. R. Hurson. Dataflow architectures and multithreading. *Computer*, 27(8):27–39, 1994.

[76] Jaejin Lee, Jun Lee, Sangmin Seo, Jungwon Kim, Seungkyun Kim, and Zehra Sura. Comic++: A software svm system for heterogeneous multicore accelerator clusters. In *HPCA*, pages 1–12, 2010.

[77] Jaejin Lee, Sangmin Seo, Chihun Kim, Junghyun Kim, Posung Chun, Zehra Sura, Jungwon Kim, and SangYong Han. Comic: a coherent shared memory interface for cell be. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 303–314, New York, NY, USA, 2008. ACM.

[78] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35:50–58, February 2002.

[79] Olivier Maquelin, Herbert H. J. Hum, and Guang R. Gao. Costs and benefits of multithreading with off-the-shelf risc processors. In *Euro-Par '95: Proceedings of the First International Euro-Par Conference on Parallel Processing*, pages 117–128, London, UK, 1995. Springer-Verlag.

[80] Pedro Marcuello, Antonio Gonza'lez, and Jordi Tubella. Speculative multithreaded processors. In *International Conference on Supercomputing*, pages 77–84, 1998.

[81] Friedemann Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2:161–175, 1987.

[82] Sally A. McKee. Reflections on the memory wall. In *Proceedings of the 1st conference on Computing frontiers*, CF '04, pages 162–, New York, NY, USA, 2004. ACM.

[83] George Michael. DDM-VMS: Data-driven multithreading virtual machine for symmetric multi-cores. Dept. of Computer Science, University of Cyprus, 2011. Undergraduate Thesis.

[84] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965.

[85] Richard Murphy. On the effects of memory latency and bandwidth on supercomputer application performance. In *Workload Characterization, 2007. IISWC 2007. IEEE 10th International Symposium on*, pages 35 –43, 2007.

[86] Shashank S. Nemawarkar and Guang R. Gao. Measurement and modeling of earth-manna multithreaded architecture. In *MASCOTS '96: Proceedings of the 4th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, page 109, Washington, DC, USA, 1996. IEEE Computer Society.

[87] Maik Nijhuis, Herbert Bos, Henri E. Bal, and Cédric Augonnet. Mapping and synchronizing streaming applications on cell processors. In *HiPEAC '09: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, pages 216–230, Berlin, Heidelberg, 2009. Springer-Verlag.

[88] Rishiyur S. Nikhil, Gregory M. Papadopoulos, and Arvind. T: a multithreaded massively parallel architecture. *SIGARCH Comput. Archit. News*, 20(2):156–167, 1992.

[89] NVidia. Nvidia's geforce 8800 graphics processor. `http://www.techreport.com/reviews/2006q4/geforce-8800/index.x?pg=1`.

[90] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. *SIGPLAN Not.*, 31(9):2–11, 1996.

[91] Petros K. Panayi, Zbigniew Chamski, Samer Arandi, George Michael, and Paraskevas Evripidou. Automatic code generation for ddm-vm in gcc using graphite: A field report. In *In Proceedings of the GROW'11 Workshop*, Chamonix, France, April 2011.

[92] Gregory M. Papadopoulos and David E. Culler. Monsoon: an explicit token-store architecture. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 82–91, New York, NY, USA, 1990. ACM.

[93] David A. Patterson and John L. Hennessy. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th ed. edition, 1990.

[94] Josep M. Pérez, Pieter Bellens, Rosa M. Badia, and Jesús Labarta. Cellss: Making it easier to program the Cell Broadband Engine processor. *IBM J. Res. Dev.*, 51(5):593–604, 2007.

[95] Chuck Pheatt. Intel threading building blocks. *J. Comput. Sci. Coll.*, 23:298–298, April 2008.

[96] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesus Labarta. Hierarchical task-based programming with starss. *Int. J. High Perform. Comput. Appl.*, 23:284–299, August 2009.

[97] Sebastian Pop, Albert Cohen, Cdric Bastoul, Sylvain Girbal, Georges andr Silber, and Nicolas Vasilache. Graphite: Polyhedral analyses and optimizations for gcc. In *In Proceedings of the 2006 GCC Developers Summit*, page 2006, 2006.

[98] Sébastian Pop, Albert Cohen, Cdric Bastoul, Sylvain Girbal, P. Jouvelot, G.-A. Silber, and N. Vasilache. GRAPHITE: Loop optimizations based on the polyhedral model for GCC. In *Proc. of the 4th GCC Developper's Summit*, pages 179–198, Ottawa, Canada, June 2006.

[99] David K. Poulsen and Pen-Chung Yew. Data prefetching and data forwarding in shared memory multiprocessors. In *Proceedings of the 1994 International Conference on Parallel Processing - Volume 02*, ICPP '94, pages 280–280, Washington, DC, USA, 1994. IEEE Computer Society.

[100] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. Distributed shared memory: concepts and systems. *Parallel Distributed Technology: Systems Applications, IEEE*, 4(2):63 –71, summer 1996.

[101] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and Jim Smith. Trace processors. *Microarchitecture, IEEE/ACM International Symposium on*, 0:138, 1997.

[102] Amir Roth and Gurindar S. Sohi. Speculative data-driven multithreading. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, page 37, Washington, DC, USA, 2001. IEEE Computer Society.

[103] Carlos A. Ruggiero and John Sargeant. Control of parallelism in the manchester dataflow machine. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 1–15, London, UK, 1987. Springer-Verlag.

[104] Shuichi Sakai, Yoshinori Yamaguchi, Kei Hiraki, Yuetsu Kodama, and Toshitsugu Yuba. An architecture of a dataflow single chip processor. *SIGARCH Comput. Archit. News*, 17(3):46–53, 1989.

[105] Scott Schneider, Jae-Seung Yeom, Benjamin Rose, John C. Linford, Adrian Sandu, and Dimitrios S. Nikolopoulos. A comparison of programming models for multiprocessors with explicitly managed memory hierarchies. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 131–140, New York, NY, USA, 2009. ACM.

[106] Wolfgang Schreiner. On Engineering a Distributed Algorithm. RISC Report Series 98-20, Research Institute for Symbolic Computation (RISC), University of Linz, Schloss Hagenberg, 4232 Hagenberg, Austria, December 1998.

[107] Ulrich Sigmund and Theo Ungerer. Identifying bottlenecks in a multithreaded superscalar microprocessor. In *In EuroPar '96 Instruction Level Parallelism, volume 1124 of LNCS*, pages 797–800, 1996.

[108] Jurij Silc, Borut Robic, and Theo Ungerer. Asynchrony in parallel computing: From dataflow to multithreading. Technical Report CSD-TR-97-4, 29, 1997.

[109] Jurij Silc, Borut Robic, and Theo Ungerer. Asynchrony in parallel computing: from dataflow to multithreading. *Parall. Distr. Comput. Practices*, 1(1):57–83, 1998.

[110] James E. Smith and Andrew R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *25 years of the international symposia on Computer architecture (selected papers)*, ISCA '98, pages 291–299, New York, NY, USA, 1998. ACM.

[111] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. *SIGARCH Comput. Archit. News*, 23(2):414–425, 1995.

[112] Kyriakos Stavrou. *The TFlux Platform: A Portable Platform for Data-Driven Multithreading on Commodity Multiprocessor Systems*. PhD thesis, Dept. of Computer Science, University of Cyprus, 2009.

[113] Kyriakos Stavrou, Marios Nikolaides, Demos Pavlou, Samer Arandi, Paraskevas Evripidou, and Pedro Trancoso. TFlux: A Portable Platform for Data-Driven Multithreading on Commodity Multicore Systems. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 25–34, Washington, DC, USA, 2008. IEEE Computer Society.

[114] Kyriakos Stavrou, Demos Pavlou, Marios Nikolaides, Panayiotis Petrides, Paraskevas Evripidou, Pedro Trancoso, Zdravko Popovic, and Roberto Giorgi. Programming abstractions and toolchain for dataflow multithreading architectures. In *ISPDC*, pages 107–114, 2009.

[115] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. Wavescalar. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 291, Washington, DC, USA, 2003. IEEE Computer Society.

[116] Steven Swanson, Andrew Putnam, Martha Mercaldi, Ken Michelson, Andrew Petersen, Andrew Schwerin, Mark Oskin, and Susan J. Eggers. Area-performance trade-offs in tiled dataflow architectures. *SIGARCH Comput. Archit. News*, 34(2):314–326, 2006.

[117] Ungerer T., Robic B., and Silc J. Multithreaded processors. *Computer*, 45(3):320–348, 2002.

[118] Robert M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. Dev.*, 11:25–33, January 1967.

[119] Pedro Trancoso, Kyriakos Stavrou, and Paraskevas Evripidou. DDMCPP: The data-driven multithreading c pre-processor. In *In Proceedings of the 11th Annual Workshop on the Interaction between Compilers and Computer Architecture (Interact-11)*, 2007.

[120] Konrad Trifunovic, Albert Cohen, David Edelsohn, Li Feng, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjodin, and Ramakrishna Upadrasta. Graphite two years after first lessons learned from real-world polyhedral compilation. In *In Proceedings of the GROW'10 Workshop*, January 2010.

[121] Jordi Tubella and Antonio González. Control speculation in multithreaded processors through dynamic loop detection. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, page 14, Washington, DC, USA, 1998. IEEE Computer Society.

[122] Nathan Tuck and Dean M. Tullsen. Initial observations of the simultaneous multithreading pentium 4 processor. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 26, Washington, DC, USA, 2003. IEEE Computer Society.

[123] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. *SIGARCH Comput. Archit. News*, 24(2):191–202, 1996.

[124] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 533–544, New York, NY, USA, 1998. ACM.

[125] Theo Ungerer and Ulrich Sigmund. Evaluating a multithreaded superscalar microprocessor versus a multiprocessor chip. In *4th PASA Workshop, Juelich, World Sc. Publ*, pages 147–159, 1996.

[126] Sriram Vajapeyam and Tulika Mitra. Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences. *SIGARCH Comput. Archit. News*, 25(2):1–12, 1997.

[127] Sriram R. Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Anil Singh, Tiju Jacob, Shailendra Jain, Vasantha Erraguntla, C. Roberts, Y. Hoskote, Nitin Borkar, and Shekhar Borkar. An 80-tile sub-100-w teraflops processor in 65-nm cmos. *Solid-State Circuits, IEEE Journal of*, 43(1):29 –41, jan. 2008.

[128] T. N. Vijaykumar and Gurindar S. Sohi. Task selection for a multiscalar processor. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 81–92, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[129] David W. Wall. Limits of instruction-level parallelism. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 176–188, New York, NY, USA, 1991. ACM.

[130] Ian Watson and John Gurd. A prototype data flow computer with token labelling. In *Proceedings of the ACM 1979 National Computer Conference*, pages 623–628, 1979.

[131] Ian Watson and John Gurd. A practical data flow computer. *Computer*, 15:51–57, February 1982.

[132] William A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.

[133] Kane Yee. Numerical Solution of Inital Boundary Value Problems Involving Maxwell's Equations in Isotropic Media. *IEEE Trans. Antennas and Propagation*, 14(3):302–307, 1966.

[134] Wen yen Lin and Jean luc Gaudiot. The design of i-structure software cache system. In *Workshop on Multithreaded Execution, Architecture and Compilation*, 1998.

Appendix A

Distributed Data Management Runtime Calls

- Data allocation
  - `address` **`dvm_all_alloc`**`(int node_id,int data_size)`

    Description

    Allocates data of *data_size* bytes in the main memory of node *node_id*. This function is *collective* i.e. all the nodes participate in calling this function. The function returns the same GAS address on all the nodes.

    Implementation

    The node with identifier *node_id* allocates the data and sends the allocated address to the rest of the nodes and then it returns the address. All the other nodes wait to receive the address and once received the address is retuned.
  - `g_address` **`dvm_alloc`**`(int data_size)`

    Description

    Allocates data of *data_size* bytes in the main memory part of the GAS on the calling node. This function is not *collective* and so it is similar to *malloc()* except that it returns a GAS address. This function has a variant that allocates aligned data (used on the DDM-VM$_c$).

    Implementation

    This function internally calls *malloc()* and then returns the resulting GAS address.
  - `int` **`dvm_free`**`(g_address *addr)`

    Description

    frees the data dynamically allocated by *dvm_all_alloc(...)* or *dvm_alloc(...)*. This function can be called *collectively* or by one node. As only the node that has the dynamically allocated data in its space will free the data.

    Implementation

    This function checks if the address refers to a location on the calling node part of the GAS and if so it calls free() otherwise it returns an error.

- Data Movement and Pointers Management

  - `int` **`dvm_move`**`(g_address* src, g_address *dest, int size)`

    Description

    Moves the data between two locations in the GAS. This is a *collective* function, but only the nodes, which the source and destination data belong to are involved in the transfer. If both the source and destination addresses occur on the same node a *memcpy* is executed.

    Implementation

    The nodes having a *node_id* corresponding with either the source or destination address invoke the send and receive services of the NIU, respectively. If both source and destination are on the same node memcpy is called.

  - `g_address` **`dvm_local_to_global`**`(int node_id,void* addr)` converts a conventional memory address into a GAS address by concatenating it with the *node_id*.
  - `void*` **`dvm_global_to_local`**`(g_address *addr)` converts the GAS address into a conventional memory address by discarding the *node_id*.
  - `int` **`dvm_is_local`**`(g_address *addr)` returns true if the GAS address refers to a location on the local node memory.
  - `int` **`dvm_get_node`**`(g_adderss *addr)` returns the *node_id* to which memory the GAS address refers.
  - `int` **`dvm_is_null`**`(g_address *addr)` returns true if the GAS address is *Null*.
  - `int` **`dvm_set_null`**`(g_address *addr)` sets the GAS address to *Null*.

- Utility Routines

  - int **`GetNodeId`**`()` returns the node_id of the local node.
  - int **`GetNodesCount`**`()` returns the total number of nodes in the system.
  - int **`GetCoreNodesId`**`(int core_id)` returns the node_id of the node to which the core with identifier core_id belongs.

For full control over the the transfer of data among the nodes, the low-level NIU send and receive calls can be used:

- `int` **`niu_send_to_node`**`(void* src,int data_size,int node_id)`
- `int` **`niu_receive_from_node`**`(void*src,int data_size,int node_id)`

Appendix B

TFlux Directives

- **Unmodified Directives**

The directives specify the start and end of the program and the DDM blocks. The directives also assign the number of cores to utilize for the execution of the threads.

Number of Kernels

#pragma ddm kernel <number>	
Definition	specifies the <number> of the cores to use for thread execution. Specifying more kernels than the number of physically available cores have a negative effect on performance
Note	Must be always defined at the beginning of the program
Generates	The specified value is passed into the DDM-VM initialization routine

Start Program

#pragma ddm startprogram	
Definition	Specifies the start of the program
Note	Variables defined before this directive are considered to be shared, unless the user defines them otherwise
Generates	DDM-VM initialization routine

End Program

#pragma ddm endprogram	
Definition	Specifies the end of the program
Generates	DDM-VM shutdown routine

Start Block

#pragma ddm block <number>	
Definition	Specifies the start of the DDM block and assigns its identifier via <number>
Note	Block identifiers must be consecutive numbers. A block can contain a maximum of 64 threads
Generates	Inlet thread that contains all the DVM_SET_THREAD_TEMPLATE of all the threads in the DDM block

End Block

#pragma ddm endblock	
Definition	Specifies the end of the DDM block
Purpose	The end of a block must occur before the start of another block
Generates	Outlet thread

- **Extended Directives**

The following directives specify the boundaries of DDM threads and the producer-consumer relationships. In addition, the directives update threads consuming initialized data & handle the context manipulations.

Start Thread

#pragma ddm thread <number> kernel (*sched_mode, sched_value*) readycount <number>

Definition	Specifies the beginning of a thread and assigns the thread identifier, the scheduling policy and its associated value and the RC
Generates	DVM_CREATE_THREAD_TEMPLATE, DVM_SET_DFP, DVM_LOOKUP, DVM_SET_REFCOUNT, DVM_THREAD_START

Optional keywords:

- **arity <*number*>** The *arity* of the thread
- **import (addr : size : flag : expression : reference-count, ...)** input DFPL information
- **export (addr : size : flag : expression : reference-count, ...)** output DFPL information
- **import_export (addr : size : flag : expression : reference-count, ...)** DFPL information of data that is used as both input & output

End Thread

#pragma ddm endthread

Definition	Specifies the end of the thread
Generates	DVM_THREAD_END, DVM_UPDATE, DVM_UPDATE_MULTIPLE, calculates the ConsumerList

Optional Keywords:

- **update (consumerID, ...)** Update the consumers with ThreadId= **consumerID**
- **update (consumerID : value, ...)** Update the consumers with ThreadId= **consumerID** and context=**value**
- **update (consumerID : value1 : value2, ...)** Update the consumer invocations with ThreadId=**consumerID** and context values ranging from **value1** to **value2**
- **cond_update (consumerID : expression, ...)** Update the consumers with identifier=**consumerID** if **expression** is TRUE
- **cond_update (consumerID : value : expression, ...)** Update the consumer invocation with ThreadId=**consumerID** and context=**value** if **expression** is TRUE
- **cond_update (consumerID : value1 : value2 : expression, ...)** Update the consumer invocations with identifier=**consumerID** and context values ranging from **value1** to **value2** if **expression** is TRUE

Update Thread

#pragma ddm update(ThreadId : *context*)

Definition	Specifies the thread to decrement its RC
Note	This directive is used typically at the beginning of a DDM-VM program to decrement the RC of threads consuming initialized data
Generates	DVM_UDATE_THREAD, DVM_UPDATE_THREAD_MULTIPLE

Variants:

- **update (Thread_Id : value1 : value2)** This variant updates a range of invocations with *context* values from **value1** to **value2**.
- **cond_update (Thread_Id : value1 : expression)** This variant performs the update if **expression** is TRUE.
- **cond_update (Thread_Id : value1 : value2 : expression)** This variant performs the multiple updates if **expression** is TRUE

Context Retrieval Operator

@context, @context.0, @context.1, @context.2	
Definition	These operators are used to access the value of the context inside the body of the thread
Note	This operator is replaced with the DVM_CONTEXT macro with the appropriate GET_CONTEXT macro based on the *arity* of the thread
Generates	GET_CONTEXT_S, GET_CONTEXT_D, GET_CONTEXT_T

Context Creation Operator

@(<number>)	replaced by	**MAKE_CONTEXT_S(<number>)**
@(<number>, <number>)	replaced by	**MAKE_CONTEXT_D(<number>,<number>)**
@(<number>,<number>,<number>)	replaced by	**MAKE_CONTEXT_T(<number>,<number>, <number>)**
Definition	These operators are used to create the *context* values	
Note	This operator is replaced with the appropriate MAKE_CONTEXT based on the *arity* of the thread	
Generates	MAKE_CONTEXT_S, MAKE_CONTEXT_D, MAKE_CONTEXT_T	

Appendix C

Monitoring and Visualization Tools

C.1 DDM Execution Events

During the execution of the DDM application the Event Tracing System captures two main types of events:

- TSU execution events
- DDM threads execution events

For every event, the event code and the start and end times are recorded. The timing of events utilizes the hardware counters supported by the underlying architecture for maximum accuracy and minimum overhead. In the case of the DDM-VM$_c$ because different timers are used in the PPE and SPEs, the timers are synchronized to unify the time-line of all the events in the system.

TSU Execution Events

The TSU execution events record the different scheduling and synchronization operations occurring in the TSU. The events codes are enumerated in Table 6.

The timing of the TSU events on the DDM-VM$_c$ utilizes the PPU hardware *Time Base (TB) Register*. The value of the register is incremented every 12.5 nanoseconds which allows attaining a high resolution and accurate timing. In the DDM-VM$_s$ implementation which ran on x86 machines we used the hardware *Time Stamp Counter*, which counts the number of cycles since reset. The meaning of cycle depends on the processor manufacturer and the family but in general its related to the internal processor clock.

Table 6: The TSU Events

Event	TSU Operation
EVENT_FIRST	Event Tracing System Initialized
EVENT_SETUP	Setup Operation (data initialization, etc.)
EVENT_PROCESS_COMMAND	TSU is processing commands in the CQ (DDM-VM$_c$)
EVENT_DECREMENT_CONSUMER	TSU is decrementing the consumers' RC
EVENT_CACHEFLOW	TSU is performing CacheFlow work
EVENT_LAST	Event Tracing System shutdown

Table 7: Thread Execution Events

Event	Thread Execution Operation
EVENT_FIRST	Event Tracing System Initialized
EVENT_SETUP	Setup operation (data initialization, etc.)
EVENT_RESET_MSG	Reset the Command Buffer, ensures CQ slots are available (DDM-VM$_c$)
EVENT_COMMIT_MSG	Send the Command Buffer to the CQ in main memory (DDM-VM$_c$)
EVENT_COMPUTATION	DDM thread execution
EVENT_NEXT_THREAD	Get the next ready thread info. Includes data prefetching in the DDM-VM$_c$
EVENT_DATA_EXPORT	Exporting produced data to main memory (DDM-VM$_c$)
EVENT_LAST	Event Tracing System shutdown

DDM Threads Execution Events

The DDM Thread Execution events record the different operations occurring during the execution of the DDM Threads on the cores. The events codes are enumerated in Table 7.

The timing of the DDM thread execution utilizes the *SPU Decrementer Register*. The value of the register is decremented every 12.5 nanoseconds (on the Cell processor in the PS3), which allows attaining a high resolution and accurate timing. In the DDM-VM$_s$, the hardware *Time Stamp Counter* is utilized .

Events Processing

The TSU events are stored in a special *EventBuffer* allocated in main memory. The thread execution events are stored in a different EventBuffer allocated per core. In the DDM-VM$_s$ the thread execution events are written directly into the per core EventBuffer. In the DDM-VM$_c$ the thread execution events are stored in a small local *EventBuffer* in the LS. When this buffer becomes full, it is automatically copied (via a DMA call) to the main memory EventBuffer pertaining to the SPE. The sizes of the EventBuffers can be adjusted in the settings of the ETS system.

When the DDM-VM application finishes execution, all the EventBuffers are processed to generate the *DDM Events Summary* log-file and the *DDM Trace* log-file.

Event Summary File

The DDM Event File (*events_summary.txt*) reports the total time and percentage (relative to the total application execution) for each event. This file is a text file that gives the user and informative overview of the execution. The file reports a summary of the TSU execution events followed by the DDM execution events of individual cores in addition to the average of all the core events percentage and other useful information. The general format of the file is depicted in Figure 93. The format shows is for the DDM-VM$_c$. The format generated by the DDM-VM$_s$ is a subset of this one since it doesn't contain the information related to the S-CacheFlow work. Moreover, instead of reporting the events duration in seconds, it is mainly performed in cycles due to frequency scaling issues on the x86 architectures as will be explained next.

Trace File and the Visualization Tool

The DDM Trace file (*events_log.ddt*) records detailed information of the events of the DDM application for both the TSU and DDM thread execution. This file is the input to the *Visualization Tool*. For every event two entries are recorded, one specifying the start time and one specifying

```
---------------- TSU Execution Event Summary ------------------

Events Timing Mode:[Accumulation|All]
Total Execution Duration:[x] seconds
Setup Time:[x] second
Total Execution Time without Setup:[x] seconds
The following is the time percentages (after subtracting the setup time)
ProcessCommand         :[x%100] [x] seconds
Decrement Cons.         :[x%100] [x] seconds
CacheFlow-Total        :[x%100] [x] seconds
CacheFlow-Xfer Data    :[x%100] [x] seconds
CacheFlow-Xfer Control :[x%100] [x] seconds
Other                  :[x%100] [x] seconds

---------------- Cores Execution Event Summary ------------------

Events Timing Mode:[Accumulation|All]
Longest Total Execution Time:[x] seconds Pertaining to Core[x]

Core:[0]
-------
Total Execution Duration:[x] seconds
Setup Time:[x] seconds
Total Execution Time without Setup:[x] seconds
The following is the time percentages (after subtracting the setup time)
GetNextThread  :[x%100] [x] seconds
Computation    :[x%100] [x] seconds
DataExport     :[x%100] [x] seconds
ResetMsg       :[x%100] [x] seconds
CommitMsg      :[x%100] [x] seconds
Other          :[x%100] [x] seconds

Avg. loopings until data info is available to issue:[x] times
CacheFlow work      :[x] seconds
Idle waiting for TSU:[x] seconds
Avg. number of pending entries upon execution      :[x] entries

Core:[1]
-------
Core:[2]
-------
……
……
Core:[N-1]

---------------- Average For All Cores ------------------

Computation     :[x%100]
GetNextThread   :[x%100]
CacheFlow       :[x%100]
MsgExchange     :[x%100]
DataExport      :[x%100]
Other           :[x%100]
```

Figure 93: Format of the Events Summary file - DDM-VM$_c$

Entry	Size in Bytes
TSU Ticks Threshold	4
Cores Tick Threshold	4
HasTSU	4
HasCores	4
Number of Cores	4
Core*0* Events Size (in bytes)	4
Core*1* Events Size (in bytes)	4
...	...
Core*N-1* Events Size (in bytes)	4
Core*0* Events Data	Core*0* Events Size
Core*1* Events Data	Core*1* Events Size
...	...
Core*N-1* Events Data	Core*N-1* Events Size
TSU Events Size	4
TSU Events Data	TSU Events Size

Table 8: DDM Trace File Format

the end time. The format of the file is depicted in Table 8. The format accommodates any change in the number of recorded events, number of cores involved and changes to the size of the events which supports adding or modifying events in the system easily.

The Visualization Tool parses the DDM Trace file and displays the events visually in a time-line fashion. This provides the programmer with a detailed account of the execution of DDM-VM applications and help the programmer to optimize the application easily.

The displayed time-line spans the entire application execution. Each event is assigned a distinct color key and plotted on the time-line according to its start time and duration. The duration is measured in the *cycles* of the counter used to record the events. Plotting the events is done by mapping the event duration into pixels (smallest visual units on the screen). In the case of the DDM-VM$_c$ every cycle is approx. 0.0125 μ seconds. In the case of the DDM-VM$_s$ this number varies according to the frequency of the processor. Figure 94 depicts a screenshot of the Visualization Tool for a DDM-VM$_c$ application.

The Visualization Tool allows the user to vary the resolution of detail that is visually presented. This is equivalent to the Zooming feature found in most visual applications. This is achieved by controlling the number of cycles that is mapped to a pixel. Using the Visualization Tool is very intuitive. Using the File/Open menu command the user selects the DDM Trace file. The user can use the Zoom In (+), Zoom Out(-) buttons to control the detail. The user can also manually control the mapping value in the Editbox. The default value is 10.0. In the case of the DDM-VM$_c$ a scale shows the length of the events in time units while in the case of the DDM-VM$_s$ this is shown in cycles.

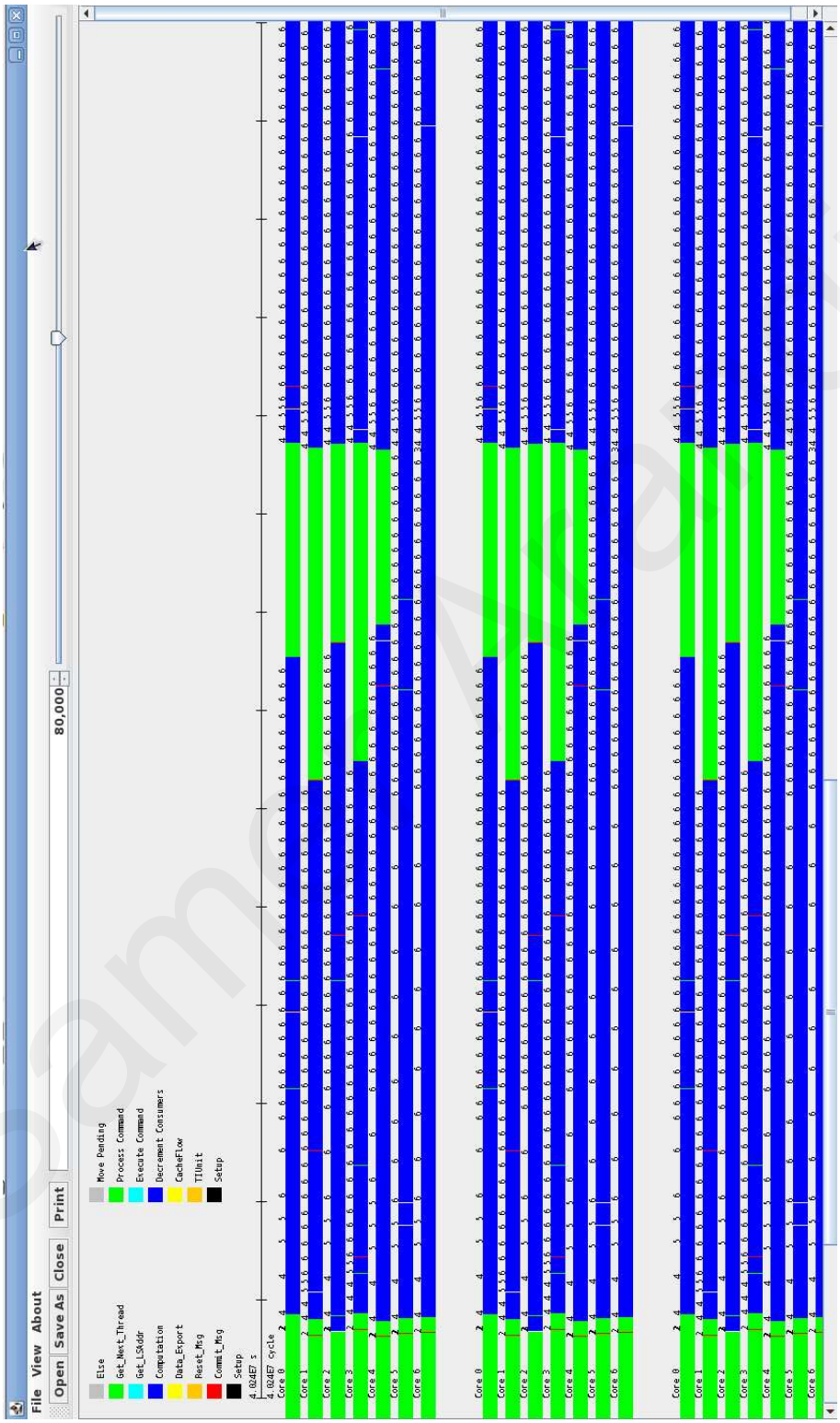


Figure 94: Visualization Tool Screenshot

C.1.1 Optimizations

Collecting information during the execution inevitably introduces overheads in any platform. In the DDM-VM the overheads are a result of two main factors: calling the timing functions and storing the event information in memory.

To address the first factor, we have used the most efficient method for timing by directly accessing the hardware timing registers available on the underlying architecture without the intervention of the Operating System. In the case of the Cell Processor the *Base Time Register* on the PPE and the *Decrementer* on the SPE provide a very accurate and stable timing interfaces. However, the timing on the DDM-VM$_s$ was less accurate as the utilized *Time Stamp Counter* is affected by the frequency scaling employed for power-saving on the *x86* multicores. Moreover, timing discrepancies existed as the timers on the different cores are not synchronized. Therefore, in the DDM-VM$_s$ log-files and visualization tools we report the timing information in terms of cycles.

The second factor was more critical since for each event we record, we write to the *Event-Buffers* in memory which affects the cache and consequently the performance. This applies to the TSU events in the DDM-VM$_s$ and all the events on the DDM-VM$_s$ as the cache hierarchy is hardware-controlled. To handle this issue we allow the program to selects between two modes of event monitoring: Detailed and Accumulating.

Monitoring Modes

In the Detailed Mode the information of every event is recorded in the *EventBuffer* in main memory as described before. In the Accumulating Mode, the timing of every event is not recorded in the *EventBuffer*, but rather accumulated in a small buffer. This almost eliminates the frequent writes to main memory, but the trace file used for the Visualization Tool is not generated. The user can select between the two modes using the ETS flags at application startup. The flags also controls enabling and disabling the various parts of the ETS to further reduce the overheads.

In the DDM-VM$_c$ implementation, handling this issue on the SPE was less problematic as the LS memory on the SPE is software-controlled and so the overheads are deterministic. Copying the events to main memory is not as frequent as in the case of the PPE since the events are first stored in a local EventBuffer and only copied when the buffer is full. However as the size of this buffer is not big (since the LS size is limited) many transfers to main memory will still occur. The main overhead stems from waiting for the DMA transfer to finish since no subsequent events can be recorded until the transfer has completed. This issue is resolved by using two local EventBuffers and applying a *double buffering* approach to avoid the waiting, which resulted in diminishing the overheads.

Although the events are recorded on different cores in the Cell processor the ETS synchronizes the start of the timing so that the reference is unified and all the different events timings are comparable.

C.2 TSU Structures Utilization and Statistics

The ETS provides information regarding the utilization of the internal TSU structures (both the common and the per-core ones) allocated in main memory. The ETS keeps track of the allocation and de-allocation of entries that occurs during applications execution. Most importantly the

```
TSU Structures Size:
---------------
Total Size of Common TSU Structures        :[x B]
Total Size of Private TSU Structures       :[x B]
Total Size of Private CacheFlow Structures :[x B]

Total Overall Structures Size              :[x B]

Shared TSU Structures Utilization
-------------------------------------------------------------------------
AQ  Max[     x] Reserved[   512]
GM  Max[     x] Reserved[   256]
CL  Max[     x] Reserved[   256]
SM  Max[     x] Reserved[   256]

Core[0] Structures Utilization:
-------------------
WQ     Max[     x] Reserved[    64]
PrioWQ Max[     x] Reserved[    32]
ExFQ   Max[     x] Reserved[    8 ]

Core[1] Structures Utilization:
-------------------
WQ     Max[     x] Reserved[    64]
PrioWQ Max[     x] Reserved[    32]
ExFQ   Max[     x] Reserved[    8 ]
……
……

Core[N-1] Structures Utilization:
-------------------
WQ     Max[     x] Reserved[    64]
PrioWQ Max[     x] Reserved[    32]
ExFQ   Max[     x] Reserved[    8 ]
```

Figure 95: Utilization File Format

number of maximum entries allocated per each TSU structure is reported. The utilization information gives insight into the performance of the different applications and the inner work of the TSU and allows us to have guidelines for selecting an optimal size for the TSU structures. The general format of the DDM utilization file (called *utilization.txt*) is depicted in Figure 95.

C.3 Supporting Distributed DDM Execution

Extending the monitoring tools to support distributed execution involves the addition of two operations. The first is synchronizing the timers across the nodes so as to have a meaningful disposition of the events of one node relative to the rest in the Visualization Tool. The synchronization occurs at the runtime initialization stage. Right after the timers on the different cores within a node are synchronized, a cross-node synchronization takes place via a simple barrier operation implemented by exchanging network messages. Once a node exists from the barrier it records the first event, which time is used later as a reference point. The rest of the Event Tracing System activities proceed without a change generating the DDM Trace file, Event Summary file and Utilization file as described previously. The only change is that the name of the file is appended with the identifier of the node.

The second operation is performed by the execution script that collects the generated files on all the nodes and stores them in one unified standard *zip* file that is saved to the *root* node. The zip file is used as input to the Visualization Tool that accesses the DDM trace files of all the nodes to display the execution events pertaining to all the nodes on the time-line. The user can also access the zip file to extract the Event Summary and Utilization files on all the nodes.