DDM-VM$_c$:The Data Driven Multithreading Virtual Machine for the Cell Processor

Samer Arandi
Department of Computer Science
University of Cyprus
Nicosia, Cyprus
samer@cs.ucy.ac.cy

Paraskevas Evripidou
Department of Computer Science
University of Cyprus
Nicosia, Cyprus
skevos@cs.ucy.ac.cy

ABSTRACT

In this paper we present the Data-Driven Multithreading Virtual Machine for the Cell Processor (DDM-VM$_c$). Data-Driven Multithreading is a non-blocking multithreading model that decouples the synchronization from the computation portions of a program allowing them to execute asynchronously in a dataflow manner. The core of the DDM model is the Thread Scheduling Unit (TSU) which schedules threads dynamically at runtime based on data availability. DDM-VM$_c$ implements the TSU as a software module running on the PPE core of the Cell, allowing the SPE cores to execute the program threads. DDM-VM$_c$ virtualizes the parallel resources of the Cell processor, handles the heterogeneity of the cores and manages the Cell memory hierarchy efficiently.

We present the architecture of DDM-VM$_c$ and provide an in-depth performance analysis using a suite of standard computational benchmarks. The evaluation shows that DDM-VM$_c$ scales well and tolerates scheduling overheads and memory latencies effectively. Furthermore, DDM-VM$_c$ compares favorably with other platforms targeting the Cell processor.

Categories and Subject Descriptors

D.1.m [**Programming Techniques**]: Miscellaneous; C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Data-flow architectures, Heterogeneous (hybrid) systems*

General Terms

Design,Languages,Performance

Keywords

multi-core, data-driven multithreading, heterogeneous

1. INTRODUCTION

The switch to multi-core has elevated *concurrency* as a major issue in utilizing the ever increasing number of cores on a single chip. Heterogeneous multi-cores make this task even harder, as different types of resources need to be individually optimized in order to achieve maximum global performance.

The Cell Broadband Engine processor [13] is a high performance heterogeneous multi-core system. It has one general purpose processor called the PPE, and eight SIMD processors, called the SPEs. Each SPE can directly operate on a small local store memory and can also access a common shared main memory through DMA calls. The memory management between the two levels must be handled explicitly by software. Cell provides a high computational power on a single chip, making it a very appealing target for high-performance applications. However, because of its heterogeneity and its novel architectural elements, programming it is not a trivial task.

The Data-Driven Multithreading Virtual Machine (DDM-VM$_c$) addresses this challenge by adopting the Data-Driven Multithreading (DDM) model of execution [10] for the Cell processor. DDM is a non-blocking multithreading model that is based on the Data-Flow model of execution[6, 2, 19]. The DDM model combines the latency tolerance and the distributed concurrency mechanisms of the Data-Flow model with the efficient execution of the sequential model. The first implementation of DDM was the Data-Driven Network of Workstations D^{2}NOW [10] which was a simulated cluster of distributed machines augmented with a hardware Thread Scheduling Unit. D^{2}NOW *CacheFlow* optimizations showed that Data-Driven scheduling could generally improve locality, contrary to the conventional wisdom at that point. The second implementation of DDM, TFlux [16], focused on portability, and thus developed a portable software platform that runs on a variety of commercial multi-core systems. TFlux also developed the first full system simulation of a DDM machine.

The DDM-VM$_c$ is a completely new implementation of DDM that differs both in focus and scope from the previous two. DDM-VM$_c$ targets a high-performance heterogeneous multi-core system that requires the programmer to handle many low-level details, such as memory management and synchronization tasks. DDM-VM$_c$ implements an efficient runtime system that provides support for scheduling, execution instantiation, synchronization, and data movement implicitly. In this context, DDM-VM$_c$ is the first heterogeneous implementation of DDM. It handles the heterogeneity by mapping the decoupled synchronization and computation tasks to the suitable core(s). DDM-VM$_c$ is also the first DDM implementation executing under constrained memory resources, which introduced a new set of challenges and emphasized the importance of exploiting locality.

DDM-VM$_c$ provides a set of C *macros* that enable the programmer to describe the parallel sections of the code and the data produced and consumed. Alternatively, the programmer can use the Concurrent Collections (CnC) [3, 4] a platform-independent, high-level parallel language with the help of a source-to-source compiler that generates the DDM-VM$_c$ program. The resulting code is compiled using the Cell SDK compilers and linked with the DDM-VM$_c$ runtime libraries.

DDM-VM$_c$ is evaluated thoroughly using a suite of standard computational benchmarks. The evaluation showed that the platform scales well and tolerates synchronization and scheduling overheads efficiently. Moreover, DDM-VM$_c$ is the first DDM implementation that can be directly compared with alternative execution models and implementations. When compared with two other platforms [18, 15] that target the Cell, DDM-VM$_c$ achieved better performance for the three computationally intensive benchmarks common to all the platforms. We believe that this is a major contribution strengthening the case that hybrid models that combine Data-Flow concurrency with efficient control-flow execution are a viable option as the basis of a new execution model for multi-core systems.

2. THE CELL HETEROGENEOUS MULTI-CORE

The Cell Broadband Engine processor (Cell B.E [13]) 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).

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.

3. DATA-DRIVEN MULTITHREADING

Data-Driven Multithreading (DDM) [10] research has combined the benefits of the Data-Flow model [6, 2, 19] in exploiting concurrency with the highly efficient sequential processing of the commodity microprocessors. Moreover, DDM can improve the locality of sequential processing by implementing deterministic data prefetching using data-driven caching policies [9]. The core of the DDM implementation is the Thread Scheduling Unit (TSU) [7] which is responsible for the scheduling of threads at run-time based on data availability.

In DDM a program consists of several threads of instructions that have producer-consumer relationships. Programming constructs such as loops and functions are mapped into DDM threads. DDM enforces single-assignment semantics across threads, and allows side-effects locally within a thread. The TSU schedules a thread for execution once 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 tuple: *ThreadID*, which is static, and *Context* which is dynamic. Each thread is paired with its *synchronization template* or *meta-data* specifying the following attributes:

1. The Instruction Frame Pointer (IPF): points to the address of the first instruction of the thread.

2. The ReadyCount (RC): a value equal to the number of producer-threads this thread needs to wait for until starting to execute.

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

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

The synchronization templates of all the threads in the DDM program constitute the *data-driven synchronization graph* which is used by the TSU for scheduling threads.

The attributes of the DDM synchronization graph are typical of any dynamic data-flow graph [2, 19] with the exception of the DFPL which is needed in our work for explicit memory management. In general, the information conveyed in the graph is sufficient to capture any structured intra-procedural control-flow. Inter-procedural extensions can also be done but were not relevant for the scope of this work.

4. DDM-VM$_C$ ARCHITECTURE

The DDM-VM$_c$ implements the DDM model on the Cell processor. The Thread Scheduling Unit (TSU) responsible for scheduling threads at run-time is implemented as a software module running primarily on the PPE core, while the execution of the threads takes place on the SPE cores.

This mapping is an efficient utilization of the Cell heterogeneous resources; as the code of the TSU that heavily uses branches and control-flow structures, is more suited to run on the general purpose PPE core originally designed for control tasks, while the threads are more suited to run on the SIMD SPE cores optimized for computational loads. The communication between the TSU and the executing threads is facilitated via DMA calls. The *Software CacheFlow* (S-CacheFlow) module in the TSU 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 1 illustrates the architecture of the DDM-VM$_c$.

The structures holding the synchronization information and the state of the TSU are allocated in main memory and shared among all the SPEs. This includes the Graph Memory (GM) which contains the synchronization templates for each thread, the Synchronization Memory (SM) which contains the *ReadyCount* values for each thread, and the Acknowledgement Queue (AQ) which holds the identification and status of threads that have finished execution. The Waiting Queue (WQ), Fire Queue (FQ) and Command Queue (CQ) hold information specific to each SPE and hence are allocated separately per SPE. The structures required for the operation of the S-CacheFlow are allocated in main memory as well. The LS memory of the SPEs holds (i) the code of the DDM threads linked with the runtime library(ii) the S-CacheFlow structures including the part of the LS which holds the data of the DDM threads, which we refer to as the *DDM Cache*.

DDM Thread Execution.

The DDM thread execution takes place on the SPEs and consists of two types of operations, computation and synchronization. The synchronization operations are performed by the runtime using simple DDM commands which are sent via a DMA call to the corresponding TSU Command Queue (CQ) in main memory. When a thread finishes execution the runtime fetches the information of the next thread to execute from the corresponding FQ in main memory via a DMA call as well.

Thread Scheduling Unit.

The TSU running on the PPE processes the commands in the CQ of every SPE. The commands either update the TSU structures or inform the TSU that the current executing thread on that SPE has finished. In the latter case, the information of the completed thread is inserted into the AQ and used to update the *ReadyCount* of the consumers of the thread that has completed execution. If any of the updated consumer threads' *ReadyCount* reaches zero, this thread is scheduled for execution on an available SPE. This is done by inserting the ready thread information into the Waiting Queue (WQ) of one of the SPEs. The thread is then processed by the S-CacheFlow module which transfers the data this thread requires to the LS of the SPE and only after that the thread is deemed ready to execute and its information is moved into the Fire Queue (FQ).

Scheduling Policy.

The DDM-VM$_c$ implements a number of scheduling policies that control the mapping of ready threads to the SPE cores. The default policy distributes the threads among the SPEs in a way that maximizes load-balancing. The other policies include *static*, *modular*, and *round-robin* policies. The DDM-VM$_c$ also supports using a *custom* policy, which gives the programmer or the compilation tools the flexibility to implement a scheduling policy based on data locality or the dependency graph of the program or any other criteria.

5. SOFTWARE CACHEFLOW

CacheFlow [9] is a cache management policy utilized with the Data-Driven Multithreading 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 [9] targeted machines with hardware caches to implicitly improve the performance of DDM execution by reducing cache misses. However, on the Cell, CacheFlow is applied in a new context, that is, to manage the Cell memory hierarchy. This is challenging because the LS is a constrained memory resource demanding efficient utilization. Moreover, the LS is software-controlled, rendering many techniques applied to preserve coherency in hardware-caches prohibitively expensive. To handle this challenge, DDM-VM$_c$ utilizes the CacheFlow policy to implement *Software CacheFlow* (S-CacheFlow): a fully automated pre-fetching software cache with variable block sizes that is extended with many optimizations like adaptive multi-buffering, data re-use and reference-counting.

S-CacheFlow Structures.

To implement S-CacheFlow on the Cell a portion of the LS memory of each SPE, usually (96-128)KB, is pre-allocated for the *DDM Cache* and divided into cache blocks. The size of the blocks can vary to match each application characteristics but must be in multiples of 128B.

The TSU has a *Cache Directory* (CD) structure for each SPE to keep track of the cache blocks state. The data of each Data Frame Pointer (DFP) of a thread is allocated at least one cache block and data instances larger than one cache block are allocated in consecutive blocks. The *Remote Cache Lookup Directory* (RCLD) allocated per-SPE, keeps track of the LS addresses where the data was allocated.

S-CacheFlow Operation.

At run-time S-CacheFlow dequeues the information of ready threads from the WQ and tries to allocate the data in the *DDM Cache* at the SPE where the thread is scheduled to run. If the allocation is successful, S-CacheFlow issues DMA calls to transfer the data from main memory to the LS by placing requests in the Proxy Command Queue of the MFC of the target SPE. Threads whose DMA calls are completed are moved into the Fire Queue (FQ) indicating they are ready for execution. To preserve coherency, S-CacheFlow writes back modified cache blocks to main memory when a thread terminates. Figure 2 illustrates the algorithm for S-CacheFlow on the Cell.

Resolving the LS address of the data for each thread (required because data belonging to different threads can be present in the LS due to pre-fetching) is performed using The RCLD. The entries of the RCLD are filled by the S-CacheFlow module in the TSU and copied to the LS of the SPE via a DMA call. The runtime on the SPEs consults the RCLD, before starting the execution of every thread, to assign the pointers that will be used to access the data. The runtime consults the RCLD again, before the thread finishes execution to write-back modified data to main memory.

Adaptive Multi-buffering/Pre-fetching.

The ability to issue non-blocking DMA calls on the Cell and check their completion asynchronously allows S-CacheFlow to issue multiple DMAs for threads with more than one DFP and/or for data belonging to more than one thread in the WQ without waiting for the transfers to complete. This allows the prefetching of the data of the threads -whenever possible- and hides the latency of the data transfers and the S-CacheFlow work with computation. Therefore, effectively achieving an automatic and transparent multi-buffering that adapts to the number of ready threads and the LS space limitation.

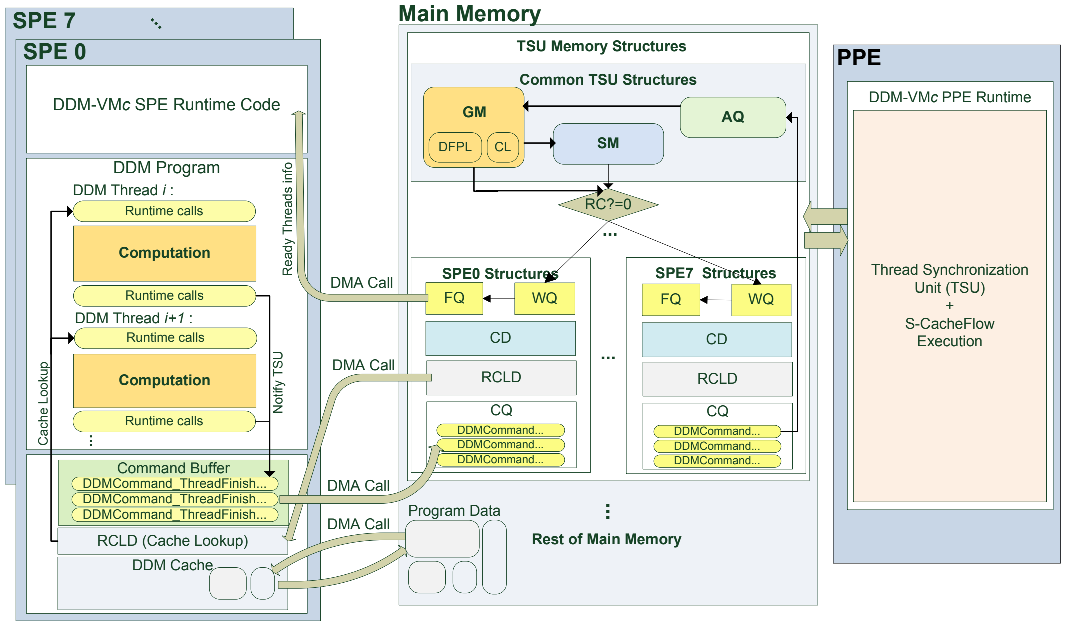


Figure 1: The Architecture of the DDM-VM$_c$

Exploiting Data Locality.

S-CacheFlow exploits data re-use, whenever more than one thread is scheduled to execute on the same SPE access the same data, by keeping the blocks of that data in the LS. The dirty bit is set for these blocks and a reference-count mechanism can be employed to decide when to write the data back to preserve coherency and avoid expensive invalidation/update operations across the SPEs. Scheduling threads that re-use data to the same SPE can be identified by the programmer or inferred from the dependency graph of the program.

Distributed S-CacheFlow.

The evaluation of the initial implementation of 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 issuing of a large number of DMAs and periodically checking their completion which overloads the PPE core that runs the TSU. 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. We call this implementation the Distributed S-CacheFlow. Evaluation of both configurations is presented in Section 7.1.

6. DDM-VM$_C$ PROGRAMMING TOOLCHAIN

The DDM-VM$_c$ utilizes the distributed synchronization mechanisms of Dynamic Dataflow as described by the U-Interpreter [2]. The program is composed of a number of re-entrant, inter-dependent DDM threads along with their *DDM Synchronization/Dependency Graph*.

The DDM-VM$_c$ programming toolchain provides programmers with three different methods to write their applications; the first is based on a set of C *macros*, the second is based on a CnC source-to-source compiler and the third is a more ambitious GCC-based auto-parallelization compiler that is still under development by our group. The resulting code of the DDM-VM$_c$ program is then compiled using the Cell SDK compilers and linked with the DDM-VM$_c$ runtime. Figure 3 shows an overview of the DDM-VM$_c$ toolchain.

DDM-VM$_c$ Macros.

This method is the most basic one where the programmer uses a set of *macros* to write the DDM-VM$_c$ program in C. The macros identify the boundaries of the threads, the data produced/consumed by the threads and the producer-consumer relationships amongst the threads. The macros expand into calls to the runtime to manage the execution of the program according to the DDM model. Programming DDM-VM$_c$ with the macros is analyzed in detail in [1].

Concurrent Collection Source-to-Source Compiler.

Concurrent Collections [3, 4] is a declarative parallel programming language, with similar semantics to DDM, which 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$_c$ programs. Thus, allowing programmers to write their applications in CnC and efficiently handling the details of the parallel execution and memory management on the Cell, which unlocks the potential of the Cell for a broader range of programmers.

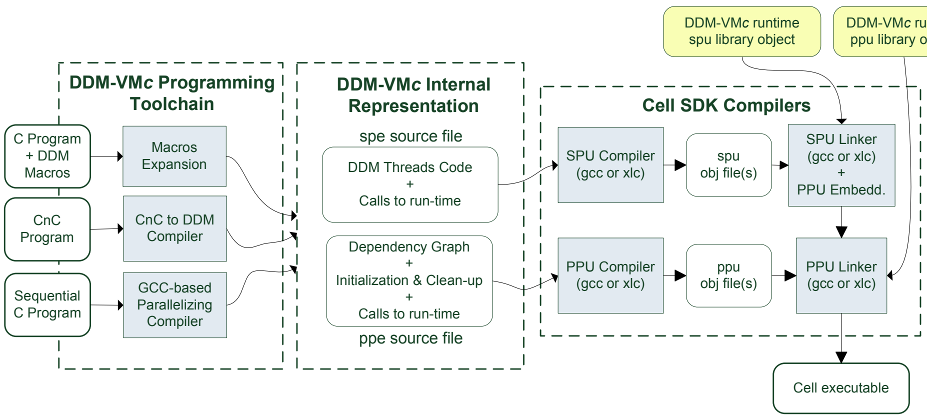
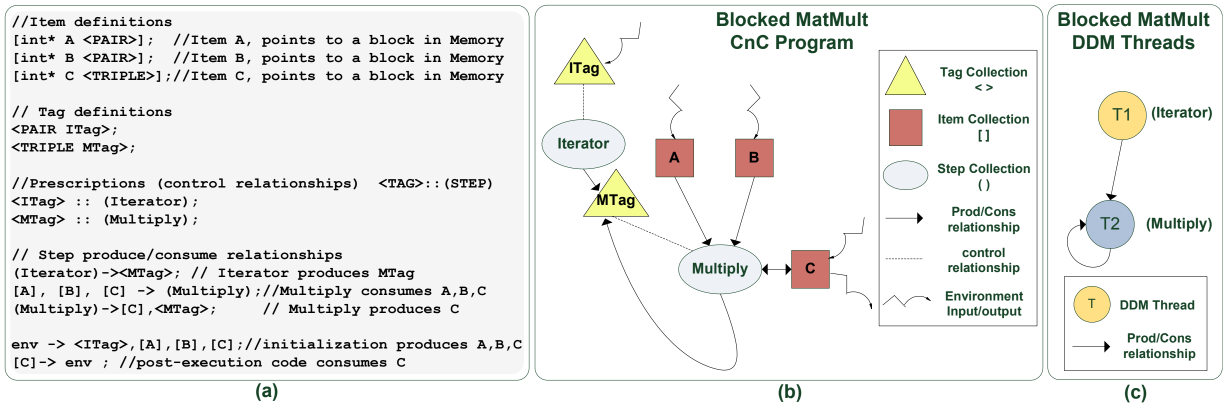

DDM-VMc runtime spu library object
DDM-VMc runtime ppu library object
DDM-VMc Programming Toolchain
DDM-VMc Internal Representation
Cell SDK Compilers
C Program + DDM Macros
Macros Expansion
CnC Program
CnC to DDM Compiler
Sequential C Program
GCC-based Parallelizing Compiler
spe source file
DDM Threads Code + Calls to run-time
Dependency Graph + Initialization & Clean-up + Calls to run-time
ppe source file
SPU Compiler (gcc or xlc)
spu obj file(s)
SPU Linker (gcc or xlc) + PPU Embedd.
PPU Compiler (gcc or xlc)
ppu obj file(s)
PPU Linker (gcc or xlc)
Cell executable

Figure 3: The DDM-VM$_c$ Programming Toolchain



```
//Item definitions
[int* A <PAIR>];  //Item A, points to a block in Memory
[int* B <PAIR>];  //Item B, points to a block in Memory
[int* C <TRIPLE>];//Item C, points to a block in Memory

// Tag definitions
<PAIR ITag>;
<TRIPLE MTag>;

//Prescriptions (control relationships)  <TAG>::(STEP)
<ITag> :: (Iterator);
<MTag> :: (Multiply);

// Step produce/consume relationships
(Iterator)-><MTag>; // Iterator produces MTag
[A], [B], [C] -> (Multiply);//Multiply consumes A,B,C
(Multiply)->[C],<MTag>;     // Multiply produces C

env -> <ITag>,[A],[B],[C];//initialization produces A,B,C
[C]-> env ; //post-execution code consumes C
```

Figure 4: 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$_c$ runtime. Figures 4-a& 4-b illustrate the textual and graphical representations of a CnC program implementing the Blocked Matrix Multiplication. The program consists of two *steps* accessing three *items*, in addition to two *tags*. Figure 4-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. The details of the mapping between CnC and DDM constructs are beyond the scope of this paper. Section 7.4 presents the preliminary evaluation results for the CnC compiler.

7. PERFORMANCE EVALUATION

A prototype of the DDM-VM$_c$ has been developed on a Sony Playstation 3 (PS3) machine with Linux 2.6.23-r1 SMP ppc64 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. The benchmarks 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 depicted at Table 1. For the benchmarks working on matrices, the matrices are non-sparse SP floating-point, except for the IDCT benchmark which works on integers.

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

7.1 Thread Granularity and S-CacheFlow Configurations

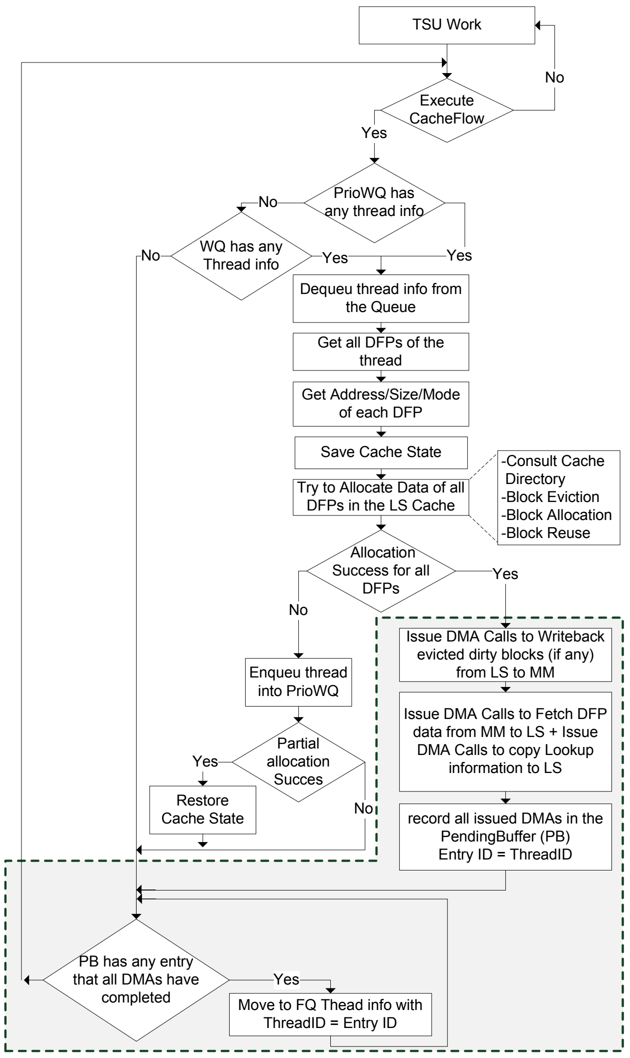
To assess the effect of thread granularity and the two S-CacheFlow configurations on performance we executed the benchmarks under both configurations. Note that different benchmarks have different thread granularities and for some of the benchmarks we have executed the same benchmark with varying thread granularities. Table 1 reports this information for every benchmark. The speedup results are depicted at Figure 5. The baseline for the speedup is the best execution out of the two configurations 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 pre-fetching/double-buffering. Applications with small granularity do not scale

Benchmark	Description	Average Granularity of Benchmark Threads		Problems Size		
		Granularity	Execution Time	Small	Medium	Large
MatMult	Blocked Matrix Multiplication	64x64 block	22.1μs	512x512	1024x1024	2048x2048
Cholesky	Blocked Cholesky Factorization	64x64 block	22μs	512x512	1024x1024	2048x2048
LU	Blocked LU Decomposition	64x64 block	1.82ms	512x512	1024x1024	2048x2048
FDTD	2D Finite Difference Time Domain [20]	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
		64x64 block	48.11μs			
IDCT	Inverse Discrete Cosine Transform	32x16 block	12.37μs	512x512	1024x1024	2048x2048
		64x32 block	49.21μs			
		64x64 block	98.8μs			
Trapez	Trapezoidal rule for integration	variable	variable	168K steps	337K steps	675K 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

Table 1: The benchmarks suite characteristics



TSU Work
Execute CacheFlow
No
Yes
PrioWQ has any thread info
No
Yes
WQ has any Thread info
No
Yes
Dequeu thread info from the Queue
Get all DFPs of the thread
Get Address/Size/Mode of each DFP
Save Cache State
Try to Allocate Data of all DFPs in the LS Cache
-Consult Cache Directory
-Block Eviction
-Block Allocation
-Block Reuse
Allocation Success for all DFPs
No
Yes
Enqueu thread into PrioWQ
Partial allocation Succes
Yes
No
Restore Cache State
Issue DMA Calls to Writeback evicted dirty blocks (if any) from LS to MM
Issue DMA Calls to Fetch DFP data from MM to LS + Issue DMA Calls to copy Lookup information to LS
record all issued DMAs in the PendingBuffer (PB) Entry ID = ThreadID
PB has any entry that all DMAs have completed
Yes
Move to FQ Thead info with ThreadID = Entry ID

Figure 2: S-CacheFlow Algorithm (shaded parts are executed on the SPE in the Distributed S-CacheFlow configuration)

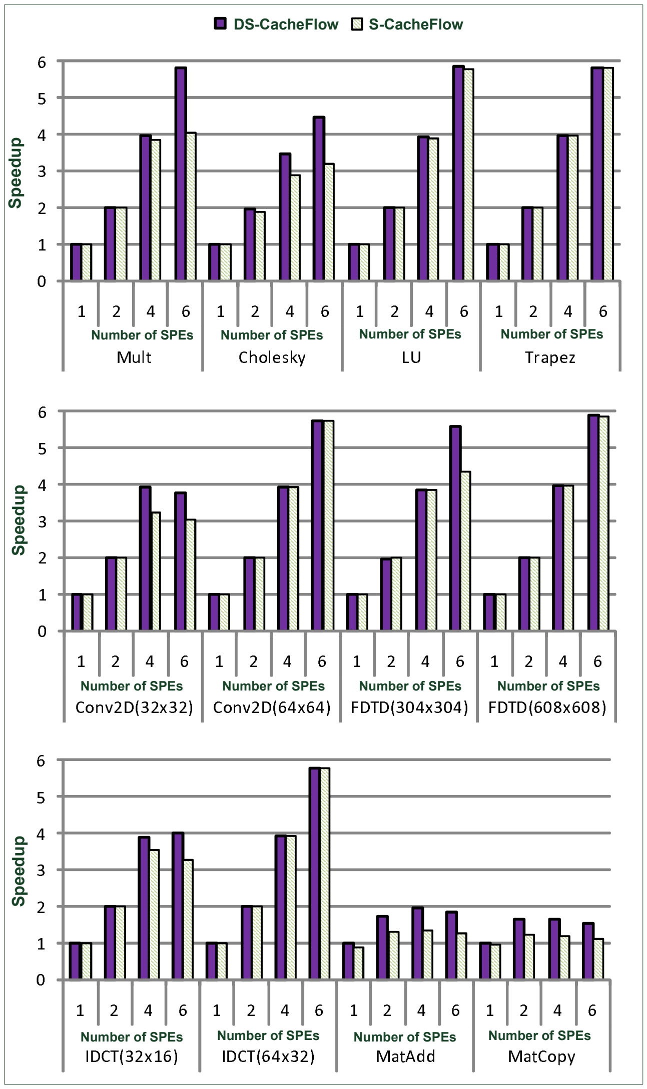


Figure 5: Effect of thread granularity and S-CacheFlow v.s. Distributed S-CacheFlow

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 v.s Distributed S-CacheFlow.

Comparing the results of the two S-CacheFlow configurations, 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 configuration is clear when the number of cores increases to 4 and higher, as previously explained in section 5. It is worthy to note that both configurations perform equally well for benchmarks that are not data-intensive (Trapez) or for ones that have a large enough granularity (ex. LU) that allows the TSU to overlap the work of scheduling and data management at higher number of cores.

Figure 6 depicts the average activities of the SPEs for the execution of MatMult under the two S-CacheFlow configurations. 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-CacheFlow because the PPE becomes a bottleneck due to the demand of the S-CacheFlow. The distributed configuration does not suffer from this and the time spent executing the computational load is kept around 90%. As such, distributed S-CacheFlow has been adopted as the default for the DDM-VM$_c$.

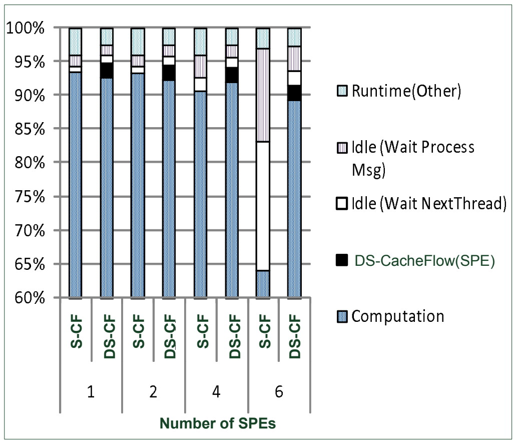


Figure 6: S-CacheFlow v.s. Distributed S-CacheFlow - MatMult SPE runtime execution activities

7.2 Concurrency and Latency Tolerance

To evaluate the potential of DDM-VM$_c$ in exploiting concurrency and tolerating synchronization and memory latencies, we have conducted a number of experiments in which we limit the number of concurrent threads 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 7 depicts the results for five of our benchmarks.

The results show that when the limit is set to one (DS-CacheFlow-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 illustrates that DDM-VM$_c$ effectively leverages the decoupling of synchronization and execution for maximum tolerance of synchronization overheads.

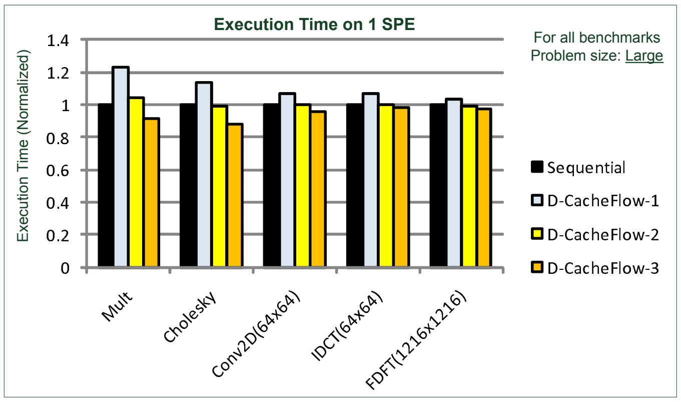


Figure 7: DDM-VM$_c$ latency tolerance

7.3 Problem Size

Figure 8 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 applications as the majority of such applications handle problems that are -at least- in the order of our "Large" problem size.

7.4 CnC Source-to-Source Compiler Preliminary Results

In this section we compare the performance of two versions of the Matrix Multiplication, one coded using the DDM *macros* v.s. one generated using a preliminary version of the CnC compiler. The results are depicted at Figure 9.

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 is currently under development. Nevertheless, we find these preliminary results for this work-in-progress very encouraging.

7.5 Overall Performance and Comparison

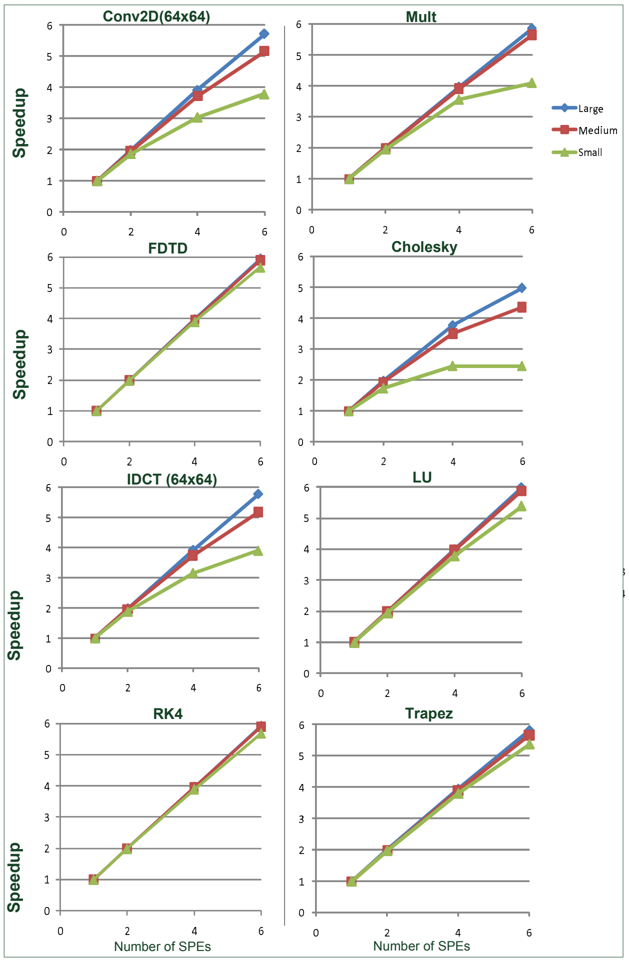


Figure 8: Effect of problem sizes on performance

In this section, we report the GFLOPs performance results of three computationally intensive applications, MatMult, Cholesky and Conv2D and compare two of them with the Sequoia[15] platform that targets the Cell processor.

The results for Sequoia were obtained by executing the MatMult and Conv2D applications found in the latest release of the Sequoia platform (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 at Figure 10 shows that the DDM-VM$_c$ performs very well achieving an average of 88% of the theoretical peak performance on MatMult, scaling almost linearly. Cholesky scales very well achieving a speedup of 5 on 6 SPEs despite its complex dependency graph. The results of the MatMult and Conv2D applications at Figure 11 also show that both applications scale almost linearly.

Comparing the results of DDM-VM$_c$ with Sequoia, DDM-VM$_c$ achieves an average of 25% and 93% performance improvement for Conv2D and MatMult, respectively. In [1] we have shown that DDM-VM$_c$ outperforms the results of CellSs [18, 14] (another platform targeting the Cell) for the MatMult and Cholesky applications. We find this as an indication of the efficiency of the DDM-VM$_c$ and its ability to perform competitively with other platforms on the Cell.

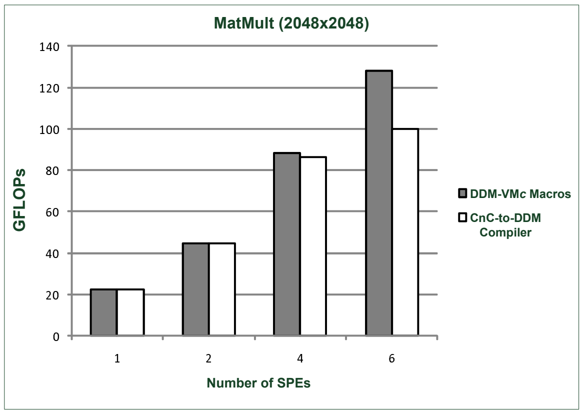


Figure 9: Performance comparison between the *macro*-coded and compiler-generated versions of the matrix multiplication program

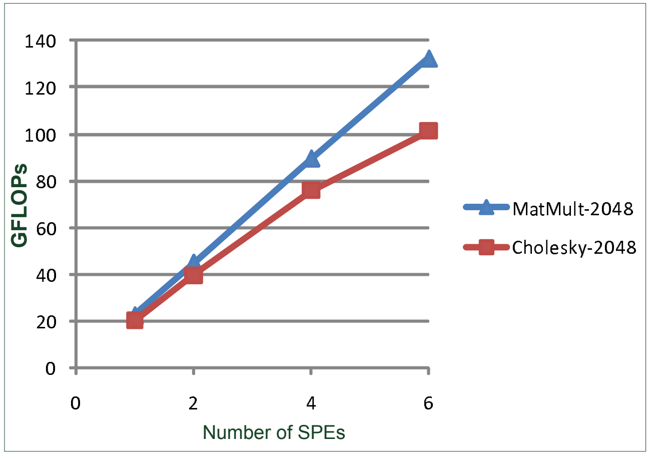


Figure 10: GFLOPs performance results for MatMult and Cholesky

8. RELATED WORK

Sequoia [15] is a programming language that facilitates the development of memory hierarchy aware parallel programs. It provides a source-to-source compiler and a run-time system for Cell. Unlike DDM-VM$_c$ Sequoia requires the use of special language constructs and types and focuses on portability. CellSs [18, 14] is a parallel programming platform available for the Cell. It schedules annotated tasks at run-time based on data-dependencies. In contrast with our model, that creates the dependency graph statically, CellSs builds it at run-time which can incurr extra overhreads. See section 7.5 for details on performance comparison.

The IBM Research Compiler targeting the Cell architecture [12] 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. Similarly, it requires the programmer to identify sections of code that can be parallelized using directives, however we believe that DDM-VM$_c$ is more general and targets problems with a higher granularity. Furthermore, our platform relies on data-flow techniques and data-flow caching policies to schedule threads and mange

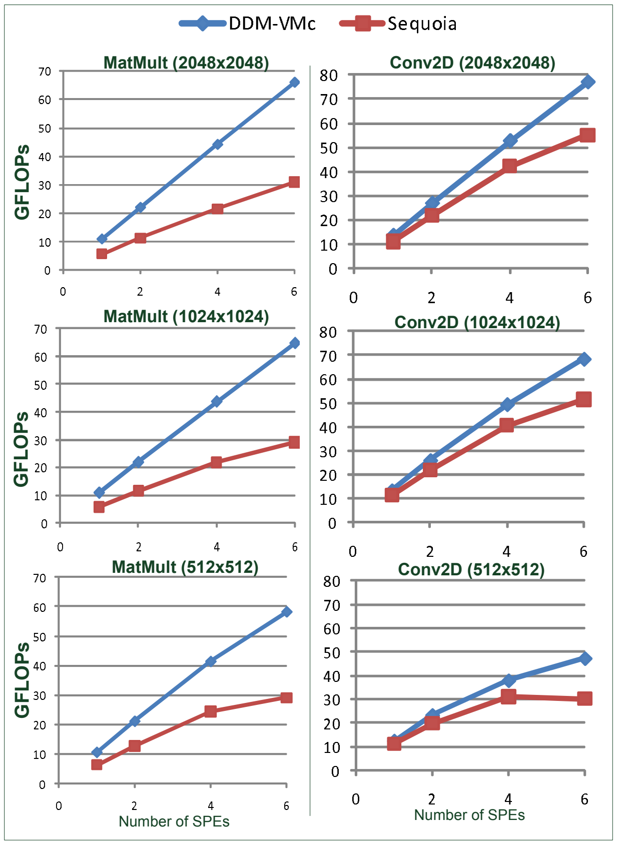


Figure 11: Comparison of DDM-VM$_c$ and Sequoia Performance for the MatMult and Conv2D applications

their data. Finally, because DDM-VM$_c$ relies for compilation on the available Cell platform compilers it can benefit from the latest optimizations and vectorization techniques provided by this compiler to optimize the code of the DDM threads running on the SPEs.

RapidMind [8] 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 that was extended to target the Cell. CellSpace[11] 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. We place DDM-VM$_c$ as a more general approach, as it doesn't require the use of any streaming abstraction and can be used for a wider range of applications.

In [12, 17] software-controlled caches are proposed to manage and optimize the tasks of data transfers on the Cell processor. In [5] direct buffering and software cache techniques are integrated to manage data transfers using both techniques in the same program. Unlike all of the aforementioned software caches which perform cache directory operations on the SPE, S-CacheFlow operations are performed on the PPE and overlapped with the execution of code on the SPEs to hide the overheads of these operations. Moreover, it enables data re-use and maintains coherency utilizing a reference-counting mechanism, thus avoiding expensive update/invalidate operations. Most notably, S-CacheFlow is utilized at the scheduling and data management levels and contains elements specific to DDM.

9. CONCLUSION AND FUTURE WORK

In this paper we presented DDM-VM$_c$, a virtual machine that implements Data-Driven Multithreading on the Cell. It utilizes Data-Flow concurrency for scheduling threads and manages data transfers automatically. Scheduling, data management and transfer operations are interleaved with the execution of threads to tolerate latencies. To develop applications, the programmer uses a set of C *macros* or the CnC language with the aid of a source-to-source compiler. The evaluation demonstrates that DDM-VM$_c$ scales well and tolerates synchronization overheads achieving very good performance and comparing favorably with other platforms.

The contributions of this work is an efficient virtual machine utilizing a completely new implementation of DDM that targets a heterogeneous high-performance multi-core system with software-managed limited memory resources. It utilizes the concept of CacheFlow for developing an automated and efficient memory management for the Cell. A distributed implementation of S-Cacheflow that supports locality has been developed through extensive analysis and experimentation. DDM-VM$_c$ is the first implementation of DDM that allows programmers to use CnC to produce DDM programs with the aid of a source-to-source compiler. DDM-VM$_c$ is also the first DDM implementation that can be directly compared with other systems. When comparing with two other platforms that target the Cell, DDM-VM$_c$ achieved better performance. This is another major contribution that 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're working on porting DDM-VM$_c$ to a cluster of Cell processors and developing a GCC-based auto-parallelizing compiler to make programming DDM-VM$_c$ easier.

10. ACKNOWLEDGMENTS

This work was supported by the Cyprus Research Promotion Foundation under Grants ΔΠ/0505/25E & ΠΕΝΕΚ/ΕΝΙΣΧ/0308/44

11. REFERENCES

[1] S. Arandi and E. Paraskevas. Programming multi-core architectures using data-flow techniques. In *SAMOS X: Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages ?–?, July 2010.

[2] Arvind and K. P. Gostelow. The U-Interpreter. *IEEE Computer*, 15(2):42–49, 1982.

[3] Z. Budimlic, A. M. Chandramowlishwaran, K. Knobe, G. N. Lowney, V. Sarkar, and L. 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.

[4] Z. Budimlic, A. M. Chandramowlishwaran, K. Knobe, G. N. Lowney, V. Sarkar, and L. Treggiari. Multi-core implementations of the concurrent collections

programming model. In *CPC '09: 14th Workshop on Compilers for Parallel Computing*. Springer, 2009.

[5] T. Chen, H. Lin, and T. 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.

[6] J. 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.

[7] P. Evripidou. Thread Synchronization Unit (TSU): A Building Block for High Performance Computers. *In: Proceedings of the International Symposium on High Perfomance Computing, Fukuoka, Japan.*, 1997.

[8] R. Inc. Cell be porting and tuning with rapidmind: A case study. *White Paper; see http://www.rapidmind.net/case-cell.php*.

[9] C. Kyriacou, P. Evripidou, and P. Trancoso. Cacheflow: A Short-Term Optimal Cache Management Policy for Data Driven Multithreading,. *Proc. EuroPar-04*, pages 561–570, Aug. 2004.

[10] C. Kyriacou, P. Evripidou, and P. Trancoso. Data-Driven Multithreading Using Conventional Microprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 17(10):1176–1188, 2006.

[11] M. Nijhuis, H. Bos, H. E. Bal, and C. 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.

[12] A. E. E. *et al.* Using advanced compiler technology to exploit the performance of the Cell Broadband EngineTM architecture. *IBM Syst. J.*, 45(1):59–84, 2006.

[13] J. A. K. *et al.* Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):561–570, 2005.

[14] J. P. P. *et al.* Cellss: Making it easier to program the Cell Broadband Engine processor. *IBM J. Res. Dev.*, 51(5):593–604, 2007.

[15] K. F. *et al.* 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.

[16] K. S. *et al.* 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.

[17] M. G. *et al.* 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.

[18] P. B. *et al.* 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] I. Watson and J. Gurd. A Practical Data Flow Computer. *IEEE Computer*, 15(2):51–57, 1982.

[20] K. 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.