Int J Parallel Prog (2018) 46:198–224
https://doi.org/10.1007/s10766-016-0486-6

Data-Driven Thread Execution on Heterogeneous Processors

Samer Arandi[1] · **George Matheou**[2] · **Costas Kyriacou**[3] · **Paraskevas Evripidou**[2]

Received: 22 August 2016 / Accepted: 29 December 2016 / Published online: 8 February 2017

Abstract In this paper we report our experience in implementing and evaluating the Data-Driven Multithreading (DDM) model on a heterogeneous multi-core processor. DDM 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. Thread dependencies are determined by the compiler/programmer while thread scheduling is done dynamically at runtime based on data availability. The target processor for this implementation is the Cell processor. We call this implementation the Data-Driven Multithreading Virtual Machine for the Cell processor (DDM-VM$_c$). Thread scheduling is handled in software by the Power Processing Element core of the Cell while the Synergistic Processing Element cores execute the program threads. DDM-VM$_c$ virtualizes the parallel resources of the Cell, handles the heterogeneity of the cores, manages the Cell memory hierarchy efficiently and supports

This work is partially supported by the Cyprus Research Promotion Foundation under Grant PENEK/ENISX/0308/44 and the EU FP7 TeraFlux project.

✉ George Matheou
geomat@cs.ucy.ac.cy

Samer Arandi
arandi@najah.edu

Costas Kyriacou
eng.kc@frederick.ac.cy

Paraskevas Evripidou
skevos@cs.ucy.ac.cy

1 Computer Engineering Department, An-Najah National University, Nablus, Palestine

2 Department of Computer Science, University of Cyprus, Nicosia, Cyprus

3 Department of Computer Science and Engineering, Frederick University, Nicosia, Cyprus

distributed execution across a cluster of Cell nodes. DDM-VM$_c$ has been implemented on a single Cell processor with six computation cores, as well as, on a four Cell processor cluster with 24 computation cores. We present an in-depth performance analysis of DDM-VM$_c$, using a suite of standard computational benchmarks. The evaluation shows that DDM-VM$_c$ scales well and tolerates scheduling overheads, memory and communication latencies effectively. Furthermore, DDM-VM$_c$ compares favorably with other platforms targeting the Cell processor, such as, the CellSs and Sequoia.

Keywords Multi-core systems · Data-Driven Multithreading · Dataflow scheduling · CacheFlow

1 Introduction

Exploiting concurrency is a major issue in utilizing the ever increasing number of cores on multi-core and many-core processors. Threaded Dataflow is proposed as a programming paradigm for exploiting concurrency and tolerating memory and synchronization latencies efficiently [5,14,20,30]. The majority of the proposed threaded dataflow systems execute the thread instructions sequentially in program order, while thread synchronization is achieved using dataflow principles.

Data Driven Multithreading (DDM) [20] is a threaded dataflow model based on the principles of dataflow execution [6,11,32]. A DDM program consists of a number of producer-consumer threads that are scheduled based on data availability. DDM threads are non-blocking threads, i.e., once they are issued for execution, they are executed to completion. The DDM model combines the latency tolerance and the distributed concurrency mechanisms of the dataflow model with the efficient execution of the sequential model. The core of the DDM model is the Thread Scheduling Unit (TSU) which is responsible for the scheduling of threads at run-time based on data availability. Several software [2,3,23,24,26,31] and hardware [21,22] implementations of the DDM model have been developed that target homogeneous and heterogeneous multi-core systems as well as distributed systems.

Recent research in heterogeneous systems has identified significant advantages of such systems over homogeneous ones in terms of power and throughput and in addressing the effects of Amdahl's law on the performance of parallel applications [18]. A heterogeneous multi-core architecture has the potential to match each application to the core best suited to meet its performance demands. Two representative heterogeneous multi-core systems are the Cell microprocessor [17] and the Parallella system [28]. However, heterogeneous multi-cores make the task of exploiting concurrency even harder, as different types of resources need to be individually optimized in order to achieve maximum global performance.

In this paper we report our experience in applying data-driven scheduling on heterogeneous multi-core processors through the Data-Driven Multithreading Virtual Machine for the Cell processor (DDM-VM$_c$) [1,3]. DDM-VM$_c$ is an implementation of the DDM model that 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. It handles the heterogeneity by mapping the decoupled synchronization and computation tasks to the suitable core(s).

To program the DDM-VMc, the developer uses a set of C *macros* that describe (i) the boundaries of the threads, (ii) the producer-consumer relationships amongst the threads (iii) and the data produced and consumed by each thread. The macros expand to calls to the runtime to manage the execution of the program according to the DDM model. In this paper we present the results of the *macro-based* approach and the initial results for one of the tools under-development: a source-to-source compiler that generates DDM-VM$_c$ programs from Concurrent Collections (CnC) [9], a platform-independent, high-level parallel language. In both approaches, the resulting code is compiled using the Cell SDK compilers and linked with the DDM-VM$_c$ runtime libraries.

DDM-VM$_c$ is evaluated using a suite of standard computational benchmarks. The comprehensive evaluation showed that the platform scales well and tolerates synchronization and scheduling overheads efficiently. Moreover, when comparing the DDM-VM$_c$ directly with two other alternative execution models on the Cell (CellSs [7] and Sequoia [13]), DDM-VM$_c$ achieves better performance for the computationally intensive benchmarks.

Finally, we present the evaluation of the distributed DDM-VM$_c$ implementation, on a cluster of four Cell processors with a total of 24 computation cores. We believe that the work presented in this paper is a major contribution strengthening the case that hybrid models, that combine dataflow concurrency with efficient control-flow execution, are a viable option as the basis of a new execution model for multi-core heterogeneous systems.

2 Background

2.1 The Cell Heterogeneous Multi-core

The Cell Broadband Engine processor (Cell/B.E) [17] is a heterogeneous multi-core chip composed of one general-purpose RISC processor called the Power Processing Element (PPE) and eight fully-functional SIMD co-processors called the Synergistic Processing Elements (SPEs) 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 256 KB 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.

2.2 Data-Driven Multithreading

Data-Driven Multithreading (DDM) [20] is a non-blocking multithreading model that combines the benefits of the dataflow model [6,11,32] 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 [19]. The core of the DDM implementation is the Thread Scheduling Unit (TSU) which is responsible for the scheduling of threads at runtime based on data availability. Scheduling based on data availability can effectively tolerate synchronization and communication latencies.

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. The TSU schedules a thread for execution once all the producers of this thread have completed their execution. This ensures that all the data needed by this thread 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.

Threads are identified by the tuple: <*ThreadID*, *Context*>. The former is a static value. The latter is a dynamic value distinguishing multiple invocations of the same thread. Each thread is paired with its *synchronization template* or *meta-data* specifying the following attributes:

1. Instruction Frame Pointer (IFP): points to the address of the first instruction of the thread.
2. Ready Count (RC): a value equal to the number of producer-threads the thread needs to wait for until starting to execute.
3. Data Frame Pointer List (DFPL): a list of pointers to the data inputs/outputs assigned for the thread.
4. Consumer List (CL): a list of the thread's consumers that is used to determine which RC 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 to schedule threads dynamically at runtime. The attributes of the DDM synchronization graph are typical of any dynamic dataflow graph [6,32] with the exception of the DFPL which is needed in our work for explicit memory management.

2.2.1 DDM Implementations

The first implementation of DDM was the Data-Driven Network of Workstations D^2NOW [20], which was a simulated cluster of distributed machines augmented with a hardware TSU. D^2NOW *CacheFlow* [19] optimizations showed that data-driven scheduling could generally improve locality, contrary to the conventional wisdom at that point. The second implementation of DDM, TFlux [31], focused on portability, and thus a portable software platform was developed that runs natively on a variety of commercial multi-core systems. Furthermore, TFlux provided a complete programming tool-chain and developed the first full system simulation of a DDM machine.

The third implementation of DDM, the Data-Driven Multithreading Virtual Machine (DDM-VM) [2,3], is a virtual machine that supports DDM execution on homogeneous and heterogeneous multi-core systems. DDM-VM supports distributed multi-core systems for both homogeneous and heterogeneous systems [26]. Further-

more, DDM-VM allows both compile-time and run-time dependency resolution [4]. The latest software implementation of DDM, FREDDO [23,24], is a C++ framework that supports efficient data-driven execution on conventional processors. FREDDO incorporates new features like recursion support [25] and simpler programming methodology to the DDM model through object-oriented programming.

In addition to the previously mentioned software implementation of DDM, DDM was also evaluated by two hardware implementations. In the first one, the TSU was implemented as a hardware peripheral in the Verilog language and it was evaluated through a Verilog-based simulation [21]. The results show that the TSU module can be implemented on an FPGA device with a moderate hardware budget. The second one [22] was a full hardware implementation with an 8-core system. A software API and a source-to-source compiler were provided for developing DDM applications. For evaluation purposes, a Xilinx ML605 Evaluation Board with a Xilinx Virtex-6 FPGA was used.

3 DDM-VM$_c$ Architecture

The Data-Driven Virtual Machine for the Cell (DDM-VM$_c$) is the DDM implementation targeting heterogeneous multi-cores with a host/accelerator organization and a software-managed memory hierarchy. The Cell Broadband Engine processor (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.

The Thread Scheduling Unit (TSU), which is responsible for scheduling threads at runtime, is implemented as a software module running primarily on the PPE core. The execution of the application threads takes place on the SPE cores. This mapping is an efficient utilization of the Cell's resources. 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. The threads are more suited to run on the SIMD SPE cores which are 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 an overview of the architecture of the DDM-VM$_c$.

3.1 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.

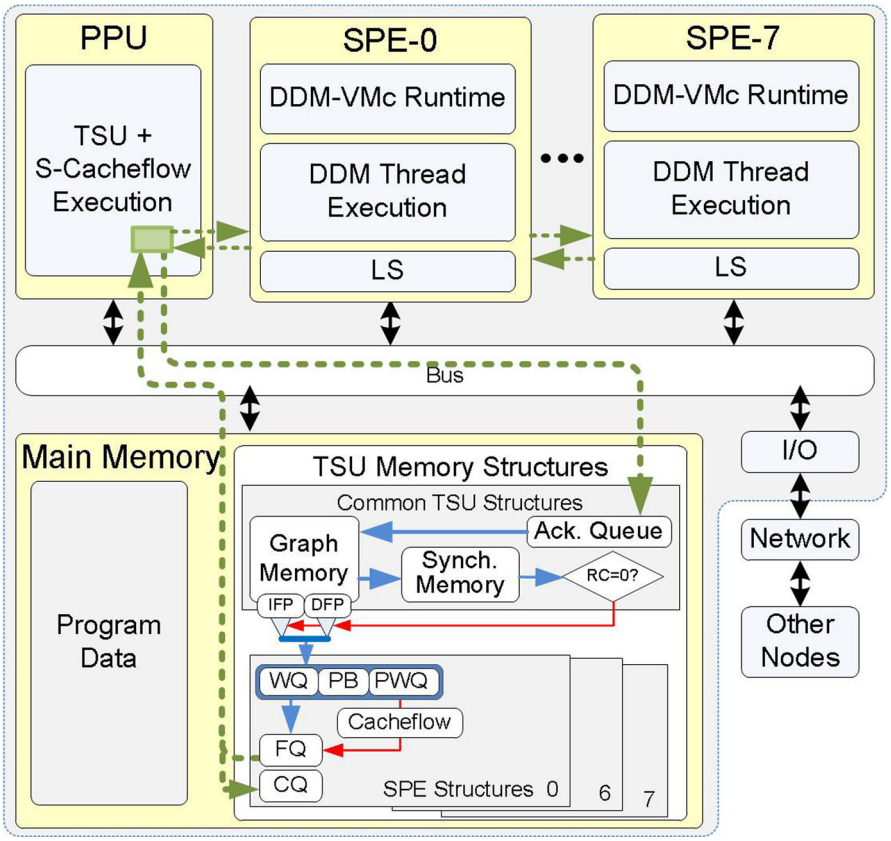


Fig. 1 The architecture of the DDM-VM$_c$

3.1.1 Common TSU Structures

Graph Memory (*GM*) holds the synchronization template of each thread. This includes: the Thread Identifier (ThreadID), the Instruction Frame Pointer (IFP), the Consumers List, the Data Frame Pointers (DFPs), the Ready Count (RC) value and the thread attributes. The thread attributes include:

- The scheduling policy and value of the thread.
- The Synchronization Memory (SM) implementation that will be used. Three different implementations are available: direct, associative and hybrid.
- A mask value that is used when the direct implementation is selected.
- The arity of the thread which specifies the loop nesting level for threads implementing loops.

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 SM. As the

performance of the SM is critical to the overall system performance, we have utilized three different implementations of the SM.

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 its execution. A request includes the consumer identifier, the Context value, an additional Context value when updating multiple invocations, and the updated value by which the consumers' RC is decremented (set to 1 by default).

3.1.2 Per-SPE TSU Structures

Command Queue (*CQ*) holds the DDM commands sent by the executing threads. These commands inform the TSU that a thread has finished its 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.

Waiting Queue (*WQ*) holds the information of threads for which the RC reached zero and are waiting for prefetching to start. This includes the ThreadID and Context value.

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.

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 this buffer is moved to the LS.

Firing Queue (*FQ*) holds the information of threads whose data has been prefetched into the LS and are ready to be executed. This includes the ThreadID, the IFP and the Context. In the distributed configuration of S-CacheFlow this queue is moved to the LS.

The LS memory of the SPEs holds (i) the code of the DDM threads linked with the runtime library and (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*.

3.2 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 its execution, the runtime fetches the information of the next thread to execute from the corresponding FQ, via a DMA call as well.

3.3 TSU Operations

The TSU running on the PPE core 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 is used to update the RC of the consumers of the thread that has completed its execution. If any of the updated consumer threads' RC reaches zero, this thread is scheduled for execution on the SPE core, selected by the scheduling policy. This is done by inserting the ready thread information into the WQ of that SPE. The thread is then processed by the S-CacheFlow module which transfers the data needed by this thread to the LS of the SPE. A thread is deemed ready to execute and its information is moved into the FQ after its data is loaded in the LS.

3.3.1 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 invocations among the SPEs in a way that maximizes load-balancing. The other implemented policies include the *static*, *round-robin*, and *modular* 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 *modular* policy uses the *Context* of the thread invocation *modulo* the number of SPE cores to select the target SPE.

The scheduling policies are assigned per-thread allowing for maximum flexibility. The DDM-VM$_c$ also supports 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.

4 Software CacheFlow (S-CacheFlow)

CacheFlow [19] is a 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 targeted machines with hardware caches where prefetching was employed to 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 processor. This is challenging because the LS is a constrained memory resource which requires efficient utilization. To handle this challenge DDM-VM$_c$ introduces the *Software CacheFlow* (S-CacheFlow): a fully automated prefetching software cache with variable cache block sizes extended with locality optimizations.

4.1 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 128 B.

The TSU has a *Cache Directory* (CD) structure for each SPE to keep track of the cache blocks state. 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. The *Remote Cache Lookup Directory* (RCLD) allocated per-SPE, keeps track of the LS addresses where the data was allocated.

4.2 S-CacheFlow Operation

At runtime, 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 FQ indicating that 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.

4.3 CacheFlow Lookup

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 prefetching) 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.

4.4 Adaptive Multi-buffering/Prefetching

The ability to issue non-blocking DMA calls on the Cell and check their completion asynchronously allows S-CacheFlow to issue multiple DMAs for the data of one or more threads 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 with the computation. Therefore, it effectively achieves an automatic and transparent multi-buffering that adapts to the number of ready threads and the LS space limitation.

4.5 Exploiting Data Locality

S-CacheFlow exploits data re-use, whenever more than one thread is scheduled to execute on the same SPE and accesses the same data, by keeping the blocks of

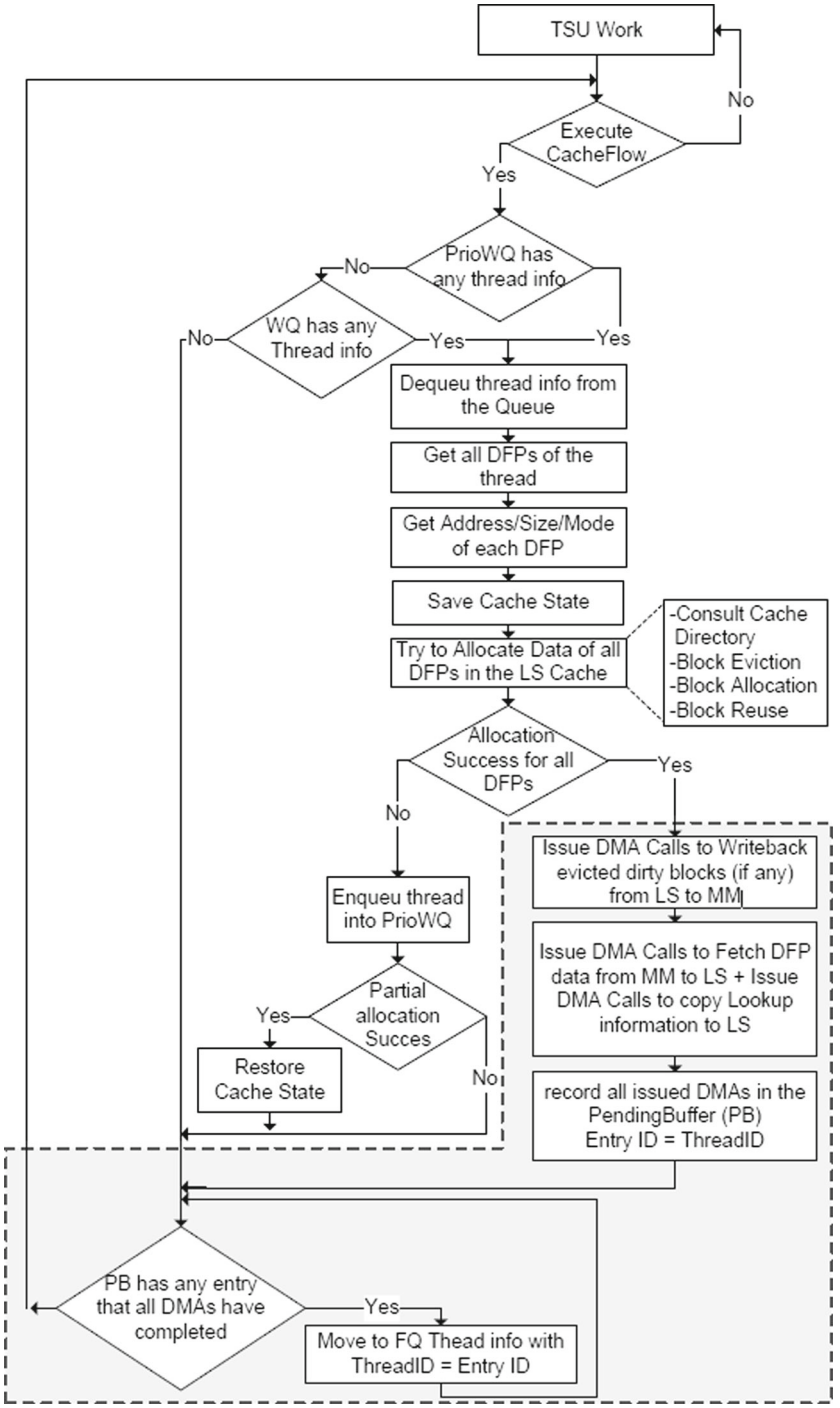


Fig. 2 S-CacheFlow algorithm (*shaded parts* are executed on the SPE in the Distributed S-CacheFlow implementation)

that data in the LS. If such data is modified, the dirty bit of its blocks is set 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 on the same SPE can be identified by the programmer or inferred from the dependency graph of the program. Performance results and more implementation details on this optimization can be found in [1].

4.6 Distributed S-CacheFlow

The evaluation of the initial implementation of S-CacheFlow scaled well for up to 4 SPE cores, but for a higher number 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 refer to this implementation as the Distributed S-CacheFlow.

5 DDM-VM$_c$ Programming Tool-chain

The DDM-VM$_c$ utilizes the distributed synchronization mechanisms of Dynamic Dataflow as described by the U-Interpreter [6]. The program is composed of a number of re-entrant, inter-dependent DDM threads along with their *DDM Synchronization/Dependency Graph*.

The success of any alternative execution model depends on the ease of programming and the efficiency of the programming approach. To this end, our group is developing a number of tools to eventually provide four alternative approaches for programming the DDM-VM$_c$:

1. *Macro-based* provides a low-level interface for programming the virtual machine. This approach has been fully implemented.
2. *TFlux preprocessor* utilizes the TFlux directives and the preprocessor tool originally developed in [31]. A subset of the directives is extended to generate the DDM-VM$_c$ macros.
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 [8,9] declarative parallel programming language to generate the DDM-VM$_c$ macros with the help of a compilation tool. An early prototype implementation is available with initial results.

The resulting code of the DDM-VM$_c$ program, which is generated by any of the approaches is compiled using the Cell SDK compilers and linked with the DDM-VM$_c$ runtime. Figure 3 shows an overview of the DDM-VM$_c$ tool-chain.

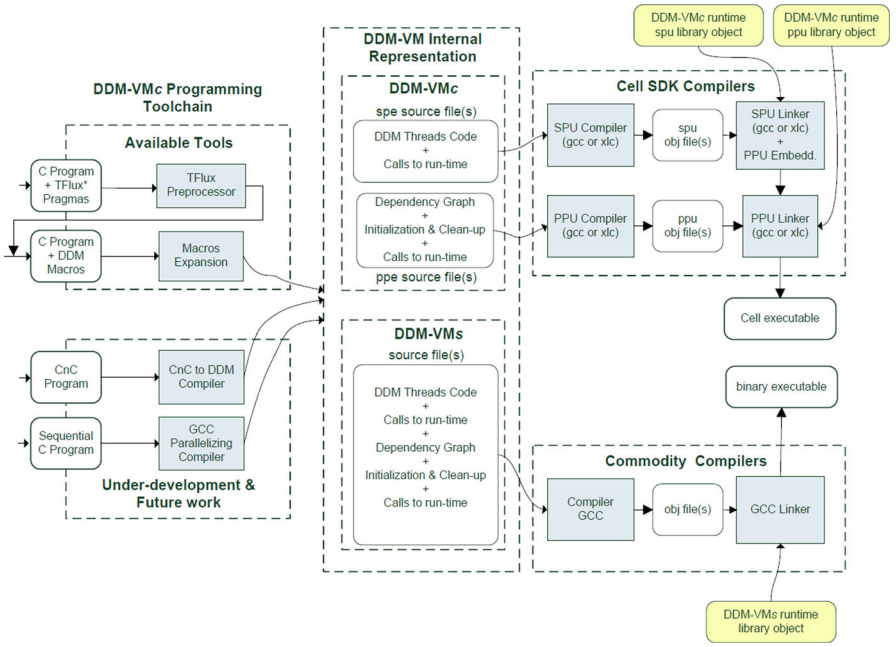


Fig. 3 The DDM-VM$_c$ programming tool-chain

5.1 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 are expanded 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 [1–3].

5.2 Concurrent Collections Source-to-Source Compiler

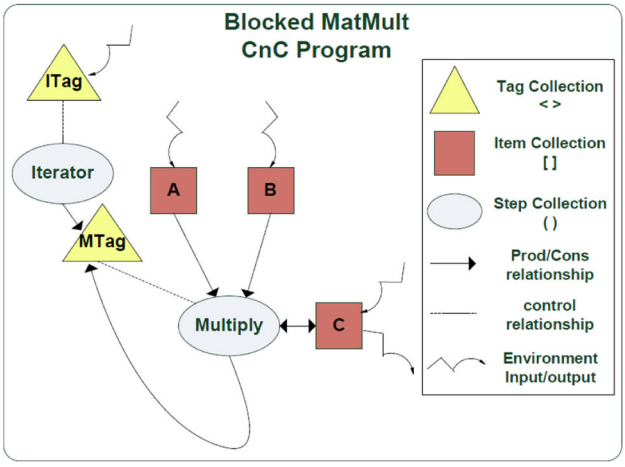
Concurrent Collections [8,9] is a declarative parallel programming language, with similar semantics to DDM, which allows programmers who lack experience in parallelism to express their parallel programs as a collection of high-level computations called *steps* that communicate via single assignment data 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.

(a)

```
//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
```

(b)



(c)

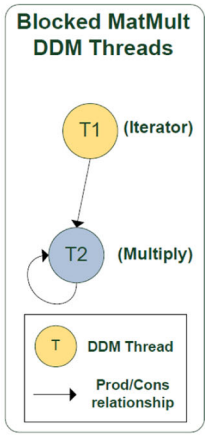


Fig. 4 The Blocked Matrix Multiplication application. **a** Textual representation of the CnC program. **b** Graphical representation of the CnC program. **c** Equivalent DDM dependency graph

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. Consequently, it unlocks the potential of the Cell for a broader range of programmers.

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. Figure 4a, 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 4c depicts the dependency graph of an 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.

6 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, the Synchronization Memory Organization and the Locality Exploitation on the performance. The second part presents a comprehensive performance evaluation which also includes a comparison with other systems targeting the Cell processor, the CellSs [7] and Sequoia [13]. In the third part we present the evaluation of the distributed DDM-VM$_c$ execution.

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. 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 Gigabit Ethernet switch with a latency of approximately 250 μs.

The benchmark suite used in the evaluation consists of nine applications featuring kernels widely used in scientific and image processing applications. The characteristics of the benchmarks are presented in Table 1. For the benchmarks working on matrices, the matrices are dense single-precision floating-point, apart from 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 a single SPE core.

6.1 TSU Evaluation

In this section we evaluate the effect on performance of the design choices related to the TSU implementation. These design choices include the resource management, the synchronization memory structures, and the locality and data re-use exploitation. 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 results of this evaluation are used as guidelines in the design choices for the implementation of the TSU.

6.1.1 Resource Management

To assess the DDM-VM$_c$ resource management control mechanisms we have executed two sets of experiments for both benchmarks. In the first, we have varied the size of the Firing Queue (FQ) and in the second, we have utilized Loop Throttling and varied the

Table 1 The benchmarks suite characteristics

Benchmark	Description	Thread Granularity	Problem Size
MatMult	Block Matrix Multiplication	64x64 Blocks	Small: 512 x 512 Medium: 1024 x 1024 Large: 2048 x 2048 Xlarge: 3072 x 3072
Cholesky	Block Cholesky Factorization (Vectorized)	64x64 Blocks	
Cholesky-S	Block Cholesky Factorization (Scalar)	64x64 Blocks	
LU	Block LU Decomposition	64x64 Blocks	
FDTD	2D Finite Difference Time Domain	304 Y-Cells	Small: 304 x 304
		608 Y-Cells	Medium: 608 x 608
		1216 Y-Cells	Large: 1216 x 1216
RK4	4^{th}-order Runge-Cutta (ODE Solver)	Variable	Small: 512 Medium: 2K Large: 3K
Conv 2D	9 x 9 Convolution Filter	32x32 Blocks	Small: 512 x 512 Medium: 1024 x 1024 Large: 2048 x 2048 Xlarge: 3072 x 3072 XXLarge: 4096 x 4096
		64x64 Blocks	
		96x96 Blocks	
IDCT	Inverse Discrete Cosine Transform	32x16 Blocks	Small: 512 x 512 Medium: 1024 x 1024 Large: 2048 x 2048 Xlarge: 3072 x 3072 XXLarge: 4096 x 4096
		64x32 Blocks	
		64x64 Blocks	
Trapez	Trapezoidal Rule for Integration	Variable	Small: 168K steps Medium: 337K steps Large: 675K steps Xlarge: 5400K steps XXLarge: 10800K steps

limit on the number of concurrent invocations of the throttled threads. To neutralize the effect of the FQ on the second set we have chosen a relatively large size for the FQ (FQ = 6). Figure 5 depicts the results.

In the first set of experiments, the results show that for both applications, as the size of the FQ 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.

TSU resource control mechanisms (e.g. setting the size of the FQ) have 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.

6.1.2 Synchronization Memory (SM) Organization

To study the effect of the Synchronization Memory implementations on the performance, we executed the two applications (Problem Size = 2048 × 2048) under three different implementations [1,2]:

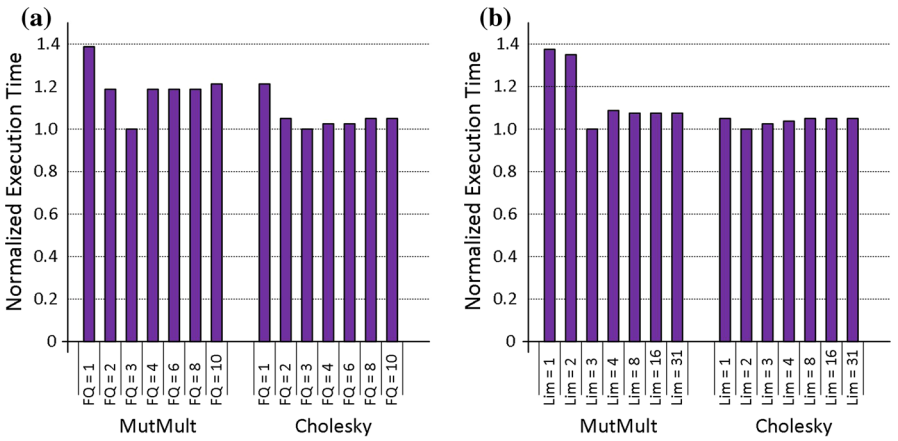


Fig. 5 Resource management control—effect of Firing Queue (FQ) size and Loop Throttling on performance. **a** Effect of varying the size of FQ, **b** effect of varying the loop bound (FQ = 6)

– *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.
– *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 deallocation 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 results are illustrated in Fig. 6. 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 second best on average. The overhead of the associative updates in this implementation 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.

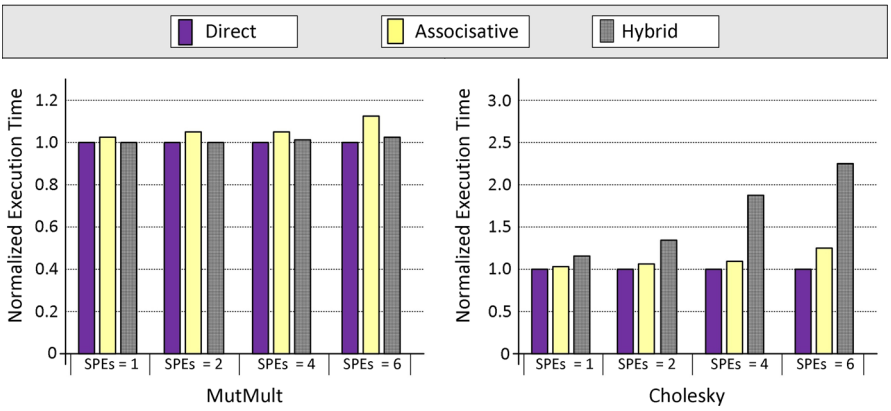


Fig. 6 Effect of the different Synchronization Memory implementations on performance

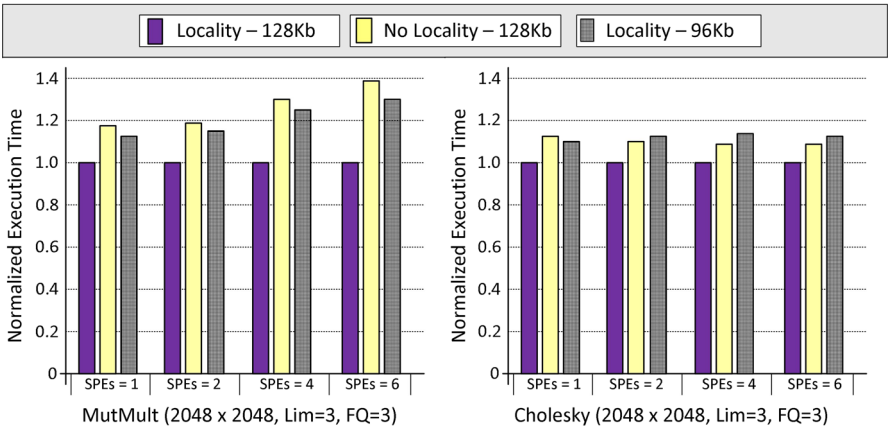


Fig. 7 Effect of locality on performance

6.1.3 Locality and Data Re-Use Exploitation

To study the effect of exploiting locality we executed the two applications with and without locality (Fig. 7). The improvement in performance when exploiting locality was achieved by simply identifying the threads that can benefit from locality by using special flags (*DATA_KEEP* and *DATA_REUSE*) in their macros.

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 three 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 the three invocations. When locality is not enabled, the data of two invocations only can fit. Fitting the data of more threads increases the probability 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 128 KB to 96 KB. This has a similar effect on the number of threads that can fit in Data Cache concurrently. The results show that the performance degrades in a fashion corresponding to the case when no locality is enabled.

The results also 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.

6.2 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 and enabled locality. Also, we have used the combination of FQ 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. Furthermore, we compare the performance of DDM-VM$_c$ with two other systems. Finally, we present the evaluation of the distributed DDM-VM$_c$ execution.

6.2.1 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 1 reports this information for every benchmark. The speedup results are depicted in Fig. 8 where 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 well when the number of SPEs increases to four and higher. This is because the TSU is doing more work 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.

S-CacheFlow Versus 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 higher. It is worthy to note that both implementations perform equally well for

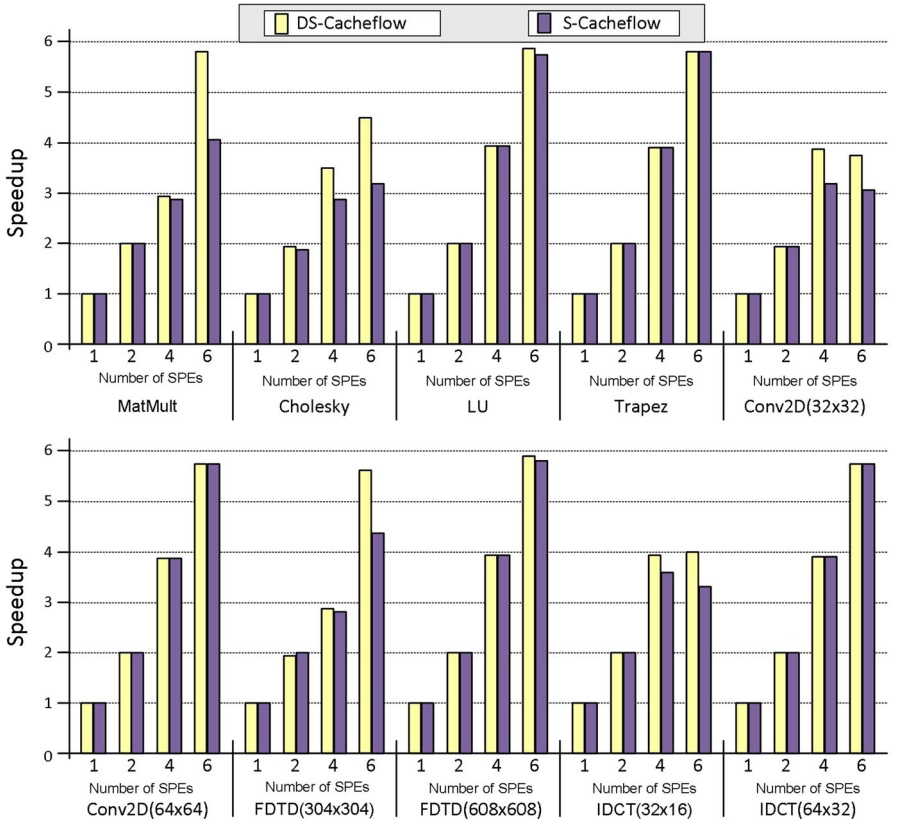


Fig. 8 Effect of thread granularity and S-CacheFlow versus Distributed S-CacheFlow

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 9 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 four SPEs, the SPEs spend more than 90% on computational work. At six SPEs, 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 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 CacheFlow implementation for the DDM-VM$_c$.

6.2.2 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* versus one generated using the preliminary version of the CnC compiler. The results are depicted in Fig. 10.

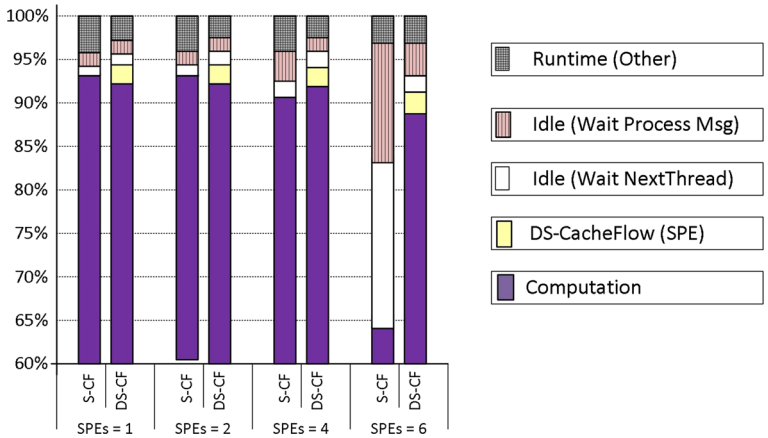


Fig. 9 S-CacheFlow versus Distributed S-CacheFlow—MatMult SPE runtime execution activities

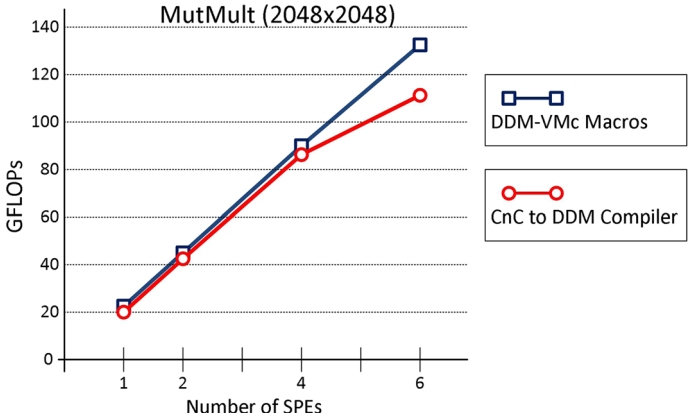


Fig. 10 Performance comparison between the *macro*-coded and the CnC compiler-generated versions of the matrix multiplication program

The evaluation shows that the compiler-generated version is performing almost the same as the macro-coded one achieving an impressive 86.5 GFLOPS for four 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.

6.2.3 Problem Size

Figure 11 depicts the results of executing eight 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. The reason for this is that the large problem sizes result in longer execution time, which amortizes initialization and parallelization overheads. We expect DDM-VM$_c$ to scale

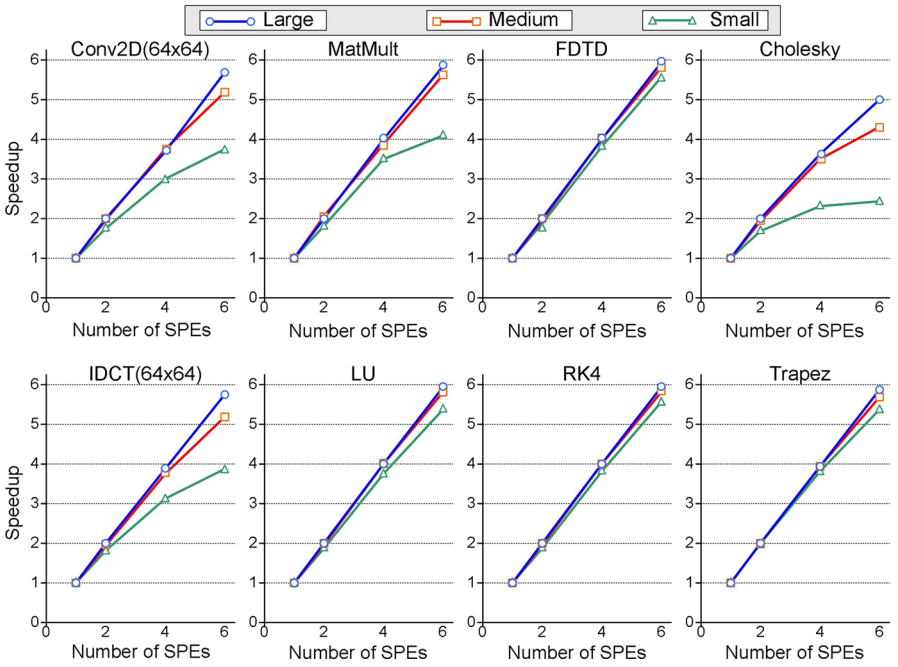


Fig. 11 Effect of problem sizes on performance

well in real life (scientific) applications since our benchmarks are representative of such applications and the typical real-world sizes employed for such applications are bigger than our *Large* problem size.

6.2.4 Overall Performance and Comparisons

In this section, we report the GFLOPs performance results of three computationally intensive applications: MatMult, Cholesky and Conv2D. We also compared our platform with two other platforms that target the Cell processor: the CellSs [7,29] (a StarSs implementation) and the Sequoia [13].

The results for CellSs were obtained by executing the MatMult and Cholesky applications found in the CellSs platform (V2.2). 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: FQ = 3, Throttling Limit = 8, Locality Enabled, Cache Size = 128 KB and *direct* SM implementation. For the Cholesky and Conv2D applications: FQ = 3, Throttling Limit = 3, Locality Enabled, Cache Size = 128 KB and *direct* SM implementation.

Figure 12 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%

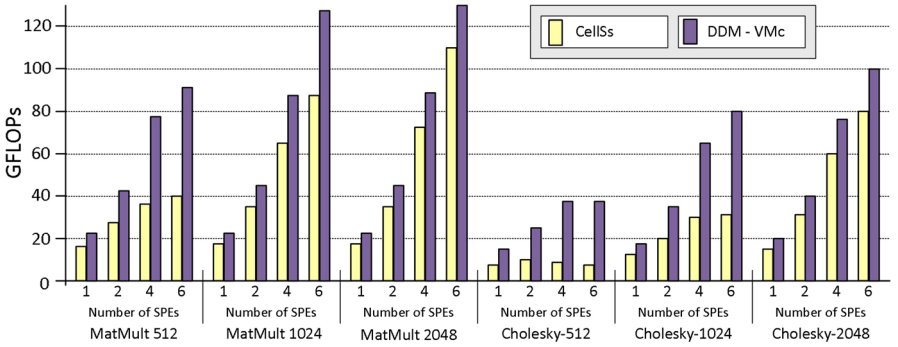


Fig. 12 Comparison of DDM-VM$_c$ and CellSs Performance for the MatMult and Cholesky applications

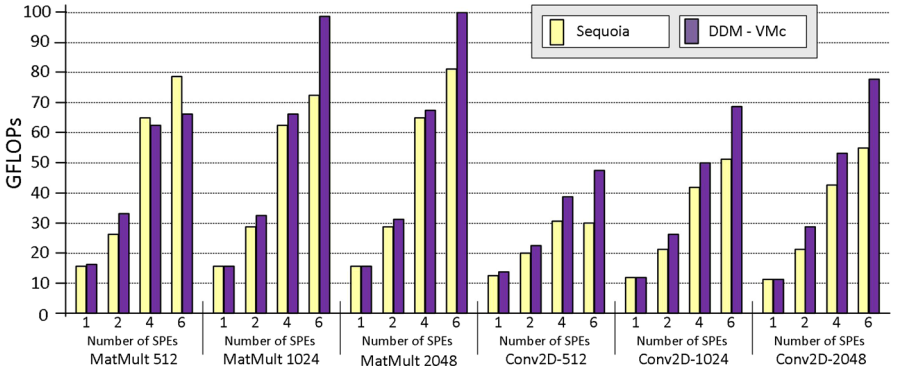


Fig. 13 Comparison of DDM-VM$_c$ and Sequoia Performance for the MatMult and Conv2D applications

for the 1024 and 512 sizes respectively. 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 Fig. 12 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 CellSs builds the dependency graph dynamically at runtime. Contrary, our model creates the dependency graph statically which introduces less overheads. 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.

Figure 13 depicts the comparison of 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 Sequoia platform (V0.9.5). 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-

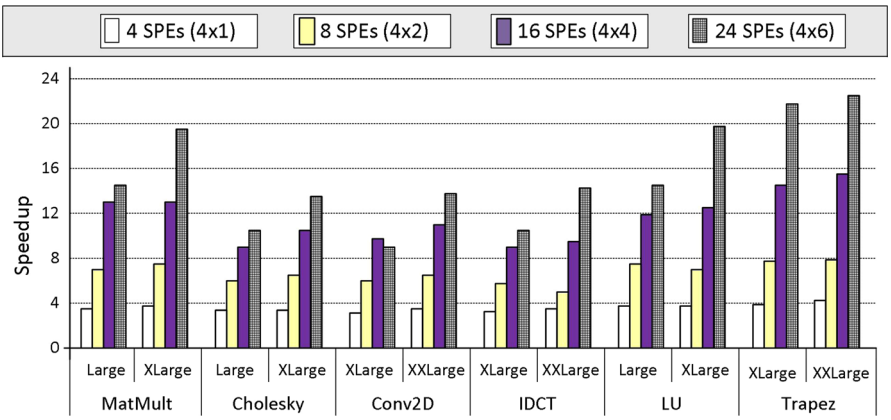


Fig. 14 Distributed DDM-VM$_c$ execution—speedup results

VM$_c$ achieves an average of 25 and 12% performance improvement for Conv2D and MatMult respectively.

To conclude, we find that these results are an indication of the efficiency of the DDM-VM$_c$ and its ability to perform favorably with other platforms on the Cell.

6.3 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 contain 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 64×64 except for the Conv2D benchmark in which we used 96×96 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 14 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.

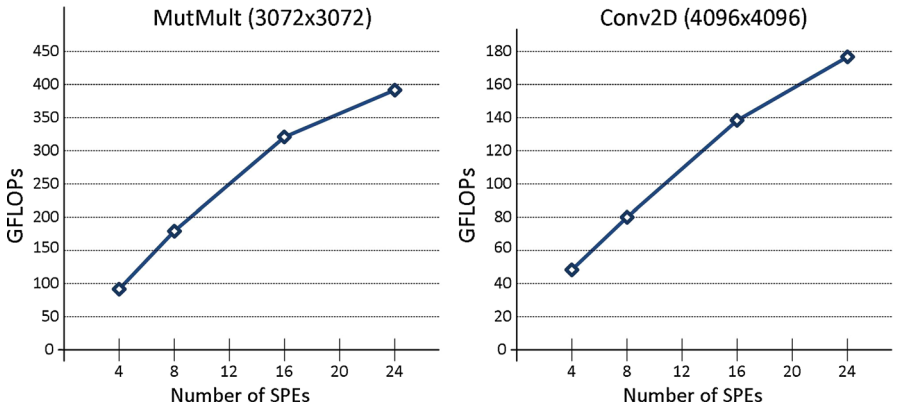


Fig. 15 GFLOPs performance results for MatMult and Conv2D

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 15 reports the GFLOPs performance results for the two computationally intensive benchmarks MatMult and Conv2D. The results illustrate that when all the SPEs are utilized on the four nodes, the system delivers an impressive 0.44 TFLOPs for the MatMult benchmark and 178 GFLOPs for the Conv2D benchmark. As such, these results demonstrate the efficiency of the distributed execution on the DDM-VM$_c$.

7 Related Work

Sequoia [13] 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 Cell. Unlike DDM-VM$_c$, Sequoia requires the use of special language constructs and types and focuses on portability.

CellSs [7,29] is a parallel programming platform available for the Cell. It schedules annotated tasks at runtime based on data-dependencies. In contrast with our model, that creates the dependency graph statically, CellSs builds it at runtime, which can incur extra overheads.

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 coarser-granularity. Furthermore, our platform relies on dataflow techniques and dataflow caching policies to schedule threads and manage their data. Finally, because DDM-VM$_c$ relies on the available Cell platform compilers, it can benefit from the latest optimizations and vectorization techniques provided by these compilers to optimize the code of the DDM threads running on the SPEs.

RapidMind [16] is a programming model that provides a set of APIs, macros and specialized data types to write streaming-like programs, which targets general multi-cores and advanced GPUs. RapidMind is also extended to target the Cell. Cell-Space [27] 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.

Eichenberger et al. [12] and Gonzàlez et al. [15] proposed software-controlled caches to manage and optimize the tasks of data transfers on the Cell processor. Chen et al. [10] integrated direct buffering and software cache techniques 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 without incurring expensive update/invalidate operations. Most notably, S-CacheFlow is utilized at the scheduling and data management levels and contains elements specific to DDM.

8 Conclusions

In this paper we presented DDM-VM$_c$, a virtual machine that implements Data-Driven Multithreading on the Cell processor. It utilizes dataflow concurrency for scheduling threads and it 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* that expand into calls to the runtime of the virtual machine. The evaluation demonstrates that DDM-VM$_c$ scales well and tolerates synchronization overheads achieving very good performance. DDM-VM$_c$ was compared with two platforms that have implementations for the Cell processor, the CellSs and Sequoia. The comparison shows that DDM-VM$_c$ outperforms both platforms.

DDM-VM$_c$ 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 DDM implementation that can be directly compared with other systems on the Cell. When comparing with two other platforms that target the Cell, DDM-VM$_c$ achieved better performance. This strengthens the case that hybrid models that combine dataflow concurrency with efficient control-flow execution are candidates for adoption as the basis of a new execution model for multi-core systems. The work on DDM has also been extended by porting DDM-VM$_c$ to a cluster of four Cell processors, with a total of 24 computation cores.

References

1. Arandi, S.: The Data-Driven Multithreading Virtual Machine. Ph.D. thesis (2012)

2. Arandi, S., Evripidou, P.: Programming multi-core architectures using data-flow techniques. In: SAMOS '10: Proceedings of the 10th International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, pp. 152–161. Samos, Greece (2010)
3. Arandi, S., Evripidou, P.: DDM-VMc: the data-driven multithreading virtual machine for the cell processor. In: Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers, HiPEAC '11, pp. 25–34. ACM, New York, NY, USA (2011). doi:10.1145/1944862.1944869
4. Arandi, S., Michael, G., Evripidou, P., Kyriacou, C.: Combining compile and run-time dependency resolution in data-driven multithreading. In: 2011 First Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM), pp. 45–52. IEEE (2011)
5. Arul, J.M., Kavi, K.M.: Scalability of scheduled data flow architecture (sdf) with register contexts. In: Proceedings Fifth International Conference on Algorithms and Architectures for Parallel Processing, 2002, pp. 214–221. IEEE (2002)
6. Arvind, Gostelow, K.P.: The u-interpreter. Computer **15**(2), 42–49 (1982). doi:10.1109/MC.1982.1653940
7. Bellens, P., Pérez, J.M., Badia, R.M., Labarta, J.: CellSs: a Programming Model for the Cell BE Architecture. In: SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, p. 86. ACM, New York, NY, USA (2006). doi:10.1145/1188455.1188546
8. Budimlic, Z., Chandramowlishwaran, A., Knobe, K., Lowney, G., Sarkar, V., Treggiari, L.: Multi-core implementations of the concurrent collections programming model. In: CPC09: 14th International Workshop on Compilers for Parallel Computers (2009)
9. Budimlic, Z., Chandramowlishwaran, A.M., Knobe, K., Lowney, G.N., Sarkar, V., Treggiari, L.: 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, pp. 47–58. ACM, New York, NY, USA (2009). doi:10.1145/1481839.1481846
10. Chen, T., Lin, H., Zhang, T.: Orchestrating data transfer for the cell/b.e. processor. In: ICS '08: Proceedings of the 22nd Annual International Conference on Supercomputing, pp. 289–298. ACM, New York, NY, USA (2008). doi:10.1145/1375527.1375570
11. Dennis, J.B.: First version of a data flow procedure language. Programming Symposium. In: Proceedings Colloque sur la Programmation, pp. 362–376. Springer, London, UK (1974)
12. Eichenberger, A.E., O'Brien, K., O'Brien, K.M., Wu, P., Chen, T., Oden, P.H., Prener, D.A., Shepherd, J.C., So, B., Sura, Z., Wang, A., Zhang, T., Zhao, P., Gschwind, M., Archambault, R., Gao, Y., Koo, R.: Using advanced compiler technology to exploit the performance of the Cell Broadband EngineTM architecture. IBM Syst. J. **45**(1), 59–84 (2006)
13. Fatahalian, K., Horn, D.R., Knight, T.J., Leem, L., Houston, M., Park, J.Y., Erez, M., Ren, M., Aiken, A., Dally, W.J., Hanrahan, P.: Sequoia: programming the memory hierarchy. In: SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, p. 83. ACM, New York, NY, USA (2006). doi:10.1145/1188455.1188543
14. Giorgi, R., Popovic, Z., Puzovic, N.: Dta-c: A decoupled multi-threaded architecture for cmp systems. In: Proceedings of IEEE SBAC-PAD (2007)
15. Gonzàlez, M., Vujic, N., Martorell, X., Ayguadé, E., Eichenberger, A.E., Chen, T., Sura, Z., Zhang, T., O'Brien, K., O'Brien, K.: 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, pp. 292–302. ACM, New York, NY, USA (2008). doi:10.1145/1454115.1454156
16. Inc., R.: Cell Be Porting and Tuning with Rapidmind: A Case Study. White Paper; see http://www.rapidmind.net/case-cell.php
17. Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D.: Introduction to the cell multiprocessor. IBM J. Res. Dev. **49**(4/5), 589–604 (2005)
18. Kumar, R., Tullsen, D.M., Jouppi, N.P., Ranganathan, P.: Heterogeneous chip multiprocessors. Computer **11**, 32–38 (2005)
19. Kyriacou, C., Evripidou, P., Trancoso, P.: Cacheflow: A short-term optimal cache management policy for data driven multithreading. In: Proceedings of EuroPar-04, pp. 561–570 (2004)
20. Kyriacou, C., Evripidou, P., Trancoso, P.: Data-driven multithreading using conventional microprocessors. IEEE Trans. Parallel Distrib. Syst. **17**(10), 1176–1188 (2006). doi:10.1109/TPDS.2006.136
21. Matheou, G., Evripidou, P.: Verilog-based simulation of hardware support for data-flow concurrency on multicore systems. In: SAMOS XIII, 2013, pp. 280–287. IEEE (2013)

22. Matheou, G., Evripidou, P.: Architectural support for data-driven execution. ACM Trans. Archit. Code Optim. **11**(4), 52:1–52:25 (2015). doi:10.1145/2686874
23. Matheou, G., Evripidou, P.: FREDDO: an efficient framework for runtime execution of data-driven objects. Technical Report TR-16-1, Department of Computer Science, University of Cyprus, Nicosia, Cyprus (2016). https://www.cs.ucy.ac.cy/docs/techreports/TR-16-1.pdf
24. Matheou, G., Evripidou, P.: FREDDO: an efficient framework for runtime execution of data-driven objects. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), p. 265 (2016)
25. Matheou, G., Watson, I., Evripidou, P.: Recursion support for the data-driven multithreading model. In: Fifth International Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM) (in press)
26. Michael, G., Arandi, S., Evripidou, P.: Data-flow concurrency on distributed multi-core systems. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), p. 515 (2013)
27. Nijhuis, M., Bos, H., Bal, H.E., Augonnet, C.: Mapping and synchronizing streaming applications on cell processors. In: HiPEAC '09: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers, pp. 216–230. Springer, Berlin, Heidelberg (2009)
28. Olofsson, A., Nordström, T., Ul-Abdin, Z.: Kickstarting high-performance energy-efficient manycore architectures with epiphany. In: 2014 48th Asilomar Conference on Signals, Systems and Computers, pp. 1719–1726. IEEE (2014)
29. Pérez, J.M., Bellens, P., Badia, R.M., Labarta, J.: Cellss: making it easier to program the Cell Broadband Engine processor. IBM J. Res. Dev. **51**(5), 593–604 (2007)
30. Solinas, M., Badia, R.M., Bodin, F., Cohen, A., Evripidou, P., Faraboschi, P., Fechner, B., Gao, G.R., Garbade, A., Girbal, S., et al.: The TERAFLUX project: exploiting the dataflow paradigm in next generation teradevices. In: Proceedings of the 2013 Euromicro Conference on Digital System Design (DSD), pp. 272–279. IEEE (2013)
31. Stavrou, K., Nikolaides, M., Pavlou, D., Arandi, S., Evripidou, P., Trancoso, P.: 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, pp. 25–34. IEEE Computer Society, Washington, DC, USA (2008). doi:10.1109/ICPP.2008.74
32. Watson, I., Gurd, J.: A practical data flow computer. Computer **15**, 51–57 (1982)