RC24931 (W1001-029) January 11, 2010
Computer Science

IBM Research Report

Mesoscale Performance Simulations of Multicore Processor Systems with Asynchronous Memory Access

Peter Altevogt
IBM STG Boeblingen

Tibor Kiss
Gamax Kft Budapest
(work done while at IBM STG Boeblingen)

Mike Kistler
IBM Research Division
Austin Research Laboratory
11501 Burnet Road
Austin, TX 78758
USA

Ram Rangan
Nvidia Corporation, Austin
(work done while at IBM Research Austin)

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Mesoscale Performance Simulations of Multicore Processor Systems with Asynchronous Memory Access

Peter Altevogt*, Tibor Kiss[$], Mike Kistler[+], Ram Rangan[#]
*IBM STG Boeblingen, [$]Gamax Kft Budapest, [+]IBM Research Austin,
[#]Nvidia Corporation Austin,

22. December 2009

Abstract

Increasing on-chip transistor densities allow for a myriad of design choices for modern multicore processors. However, conducting a meaningful design space exploration of large systems with detailed cycle-accurate simulations using large, complex workloads can be very time-consuming and can adversely impact product schedules. This is due to three main reasons: 1) the high (human) cost of developing cycle-accurate simulators, 2) long simulation times for any sufficiently detailed simulator of a large, complex system, and 3) long running times for modern workloads. While statistical sampling techniques address workload run lengths, there exists no proven technique to replace the use of detailed cycle-accurate simulators for design space exploration.

In this paper, we introduce *mesoscale simulation*, which is a methodology for design space exploration that mitigates the cost of cycle-accurate simulators. Mesoscale simulation is a hybrid approach that combines elements of high-level modeling and low-level cycle-accurate simulators to enable the construction of fast, high-fidelity performance models. Such models can be used to quickly explore vast areas of the design space and prune it down to manageable levels for cycle-accurate simulator based studies. We describe a proof-of-concept mesoscale implementation of the memory subsystem of the Cell/B.E. processor and discuss results from running various workloads.

$ - This work was done while the author was employed by IBM STG Boeblingen.
- This work was done while the author was employed by IBM Research Austin.

Table of Contents

1 Introduction

To develop a balanced design tailored to the needs of a target market, computer architects must increasingly rely on detailed simulations of processor microarchitectures, and less on intuition and experience, to understand various performance, power, and cost tradeoffs. The continuing exponential growth of on-chip transistor densities allows for many design choices. Simulations provide architects with reasonably accurate performance metrics and help them make well-informed design decisions.

Detailed performance simulations of future systems, while indispensable, pose interesting, hard-to-overcome challenges. The efficacy of a simulation environment can be measured along three axes: accuracy, speed, and cost. Accuracy of a simulation refers to how close the performance predictions are to a reference system (typically, native hardware or a very detailed simulator). Speed of simulation is simply measured by the actual wall clock time to successfully run a workload on a target simulated system, given some fixed amount of computational resources. Cost refers to both the human cost of developing detailed simulators as well as the computational resources required to run them. Ideally, a simulation environment would have high accuracy, high speed, and low cost. However, this is seldom achieved in practice. In fact, optimizing for two of the three axes, almost always results in a poor score on the third axis. The main challenge with performance simulations is to get as close as possible to the ideal on all three axes.

Prior research has used two predominant approaches to attack the broad problem of improving simulation efficiency. The first approach has focused on improving the speed and lowering the cost of developing highly accurate, detailed performance simulators [1] [2] [3]. The second approach seeks to reduce the duration of detailed simulation of long-running workloads. Techniques include using reduced input sets [4], statistically sampling portions of a workload [5] [6], a combination of checkpointing and sampling [7], and fast-forwarding an arbitrary number of instructions to quickly reach the "steady state" of a workload. The common underlying assumption for both the approaches is that fully detailed simulation is the only way to obtain highly accurate results. Consequently, such techniques suffer from poor simulation speed, have high development costs, or fall short of accurately predicting performance. Few alternative simulation paradigms exist that provide a satisfactory balance among the three axes.

One such alternative, routinely used by performance analysis teams in industry, is execution-driven or trace-driven simulation of a specific processor subsystem (for e.g., [10]) for unit-level analysis. Such unit-level models interface with "stub" models of other subsystems. Such stub models are light-weight and very often, model all incoming requests with a constant latency logic, i.e. not taking into account the feedback from the

subsystem. While such a paradigm keeps development costs low and achieves high simulation speeds by obviating the need to model and evaluate the full system in detail, the use of ad-hoc stub models sacrifices accuracy. For example, an analysis of a memory subsystem with a stub model of a processor core can be inaccurate and result in erroneous conclusions [8].

Despite issues with accuracy, there is still intrinsic value in subsystem-centric simulations for performance analysis and design space exploration. Since design improvements across processor generations typically tend to be evolutionary, it should be possible to accurately predict the performance impact of a subsystem without having to simulate the whole system.

This paper presents a new approach to subsystem-centric simulation that maintains the benefits of lower development cost and execution time, but improves the accuracy of the predicted results. Our approach captures and separates the performance behavior of fixed or unmodified components of the system, and then uses a novel simulation design to combine this performance behavior with detailed simulations of the subsystems being explored to produce performance predictions for the complete system. We show how to extract and reuse key timing information from timing traces of a whole system execution, which can be collected either from whole system simulators or tracing facilities in real hardware. Timing information thus captured enables us to greatly improve the accuracy of stub models, while retaining the speed and development cost advantages of subsystem-level simulations.

The following are the major contributions of this paper.

1. We describe our new approach to subsystem-centric simulation called *mesoscale simulation*. The subsystem being studied in detail is called the *System Under Investigation* (SUI), while the remaining subsystems are abstracted away in the *Adaptive Workload Generators* (AWG). With the proper abstraction and terminology, we show how mesoscale simulation can find application well outside the domain of modern computer systems.

2. We present a simple technique called trace distillation to isolate the timing effects of individual subsystems from whole system timing traces. The AWG uses distilled traces to enable accurate modeling of the interaction between the SUI and the rest of the system.

3. We demonstrate the accuracy, speed, and cost advantages of mesoscale simulation by comparing a mesoscale model of the Cell/B.E. against a highly detailed whole system simulator and native hardware.

The outline of this publication is as follows. We first provide an overview of related work. Then we introduce the mesoscale performance simulation methodology combining concepts from high-level performance modeling with concepts known from cycle-accurate performance simulations frequently applied in the domain of processor design[1]. We then show how to apply this mesoscale methodology to simulate the asynchronous memory access of the Cell/B.E., a hybrid multicore processor system.

2 Related Work

In this section we discuss in detail various performance simulation methodologies and compare them with our approach.

B^2SIM is a hybrid simulator for computer systems that separates execution time into cycles consumed by the core and cycles consumed in the memory hierarchy [9]. This separation then allows core cycles to be determined once for each basic block of the program using a cycle-accurate pipeline simulator. Subsequent executions of the basic block still employ the memory system model to determine memory cycles, but bypass the pipeline simulation of individual instructions and simply substitute the core cycles already measured for the basic block. This design provides speedups of up to 4x compared to a full simulation using the pipeline simulator with less than 1.25% error in reported CPI. Our work differs from B^2SIM in several significant aspects. First, we develop a general strategy for hybrid simulation and describe how it can be applied to a wide range of simulation problems, though we use computer system simulation as our primary example. Second, the processor architecture in our example system allows us to clearly distinguish time spent in the processor core from time consumed by the memory subsystem, and thus completely separate the process of determining these quantities without relying on heuristics or statistical methods. Finally, our simulation technique improves performance by more than two orders of magnitude over a full cycle-accurate

[1] Throughout this publication the terms “performance modeling” and “performance simulation” including variations of these terms have the same meaning.

simulation, while still producing highly accurate results compared both to simulations and actual hardware performance measurements.

The mlcache simulator was developed to enable exploration of novel memory hierarchy designs earlier in the design cycle [10]. Mlcache contains high-level models of the key performance-related behaviors of multi-level caches, which can then be combined in a variety of ways to easily model a wide range of memory hierarchy designs. The simulator is driven by instruction traces, which are typically collected from current systems. Mlcache is combined with a cycle-accurate instruction simulator which processes the instruction stream to determine the contribution of the processor cores to overall execution time. The accuracy of mlcache was described only in comparison to other models which were known to be incomplete, and no results were reported for performance of the simulator itself. Our work shares the goal of early memory system exploration with mlcache, but differs in several important ways. While both simulations are trace-driven, we distill the trace file beforehand, significantly increasing the speed of the subsequent simulation. We have also taken a more general approach to the design of the memory hierarchy – our SUI model is developed by hand and not a composition of prebuilt models. This may require a somewhat larger development effort, but admits a much larger range of potential designs.

The MPI Application Replay network Simulator (MARS) was developed to allow simulation of large high-performance computer systems with thousands of nodes [11]. The main focus of MARS is on network communications, but has many similarities with our mesoscale simulation. In our mesoscale simulation terminology, the SUI of a Mars simulation is the network, consisting of network adapters, links, and switches with various topologies and characteristics, and the AWG simulates the processing and network requests of individual MPI tasks running on the multi-processor nodes of the system. The processing performed at a particular node is modeled abstractly as computational delays occurring in between MPI requests. MARS captures causal relationships between MPI requests and ensures that these are maintained in the simulation. The key strengths of MARS are its detailed network model and its ability to simulate large system configurations. This is aided by the support for parallel simulations and a very high level modeling of the computing nodes based using just a few delays which can be adjusted by the user. Furthermore, it is very flexible due to the nature of the underlying modeling framework. Simulation speed is not reported explicitly, but successful simulations have been performed on configurations of up to 65,536 nodes. No results are given regarding accuracy of results. Our mesoscale simulation work differs from MARS in that it focuses on the processor, rather than the network, but more fundamentally in our method for distilling traces from real workloads in a way that preserves the affect of computational delays on request generation in detail and the affect of SUI feedback on the workload. This careful separation of effects has allowed us to

achieve very accurate results in comparison to performance measurements on actual hardware.

Paraver, and its associated simulation tool, Dinemas, are another example of a specific application of mesoscale simulation [12]. These tools are focused mainly on network communication, rather than the memory subsystem, and use an approach very similar to the MARS MPI replay engine. Applications are represented abstractly, as a trace of computation and communication operations. Dinemas simulates the application execution on a collection of nodes with a user-specified network architecture. Paraver is a visualization tool that presents the output of Dinemas in a form that facilitates analysis of communication performance. Our work develops a general approach to simulations of this type, and illustrates how it can be applied in a way that achieves high simulation accuracy and high simulation speed with modest investments in model development and simulation resources.

The simulator described by Prete et al. [13] is an early example of hybrid simulation of memory subsystems. This simulator could be driven by either an instruction trace from an actual system or by a synthetic workload generator. Processor cores are modeled abstractly as performing some computation and generating a stream of memory requests, while the memory subsystem is modeled in detail. The parameters for a synthetic workload can be extracted from an actual trace file using a process similar to our trace distillation technique. The focus of this work was on coherence protocols, and thus the organization of the memory hierarchy was fixed. While there are some similarities to our mesoscale simulation methodology, this work does not address the issues of simulation accuracy or speed of model development and execution. Furthermore, we develop a general terminology and methodology for hybrid simulation and then present our work on memory subsystem simulation as an example of this methodology.

Wild et al. [14] describe an approach for simulating System-on-Chip (SOC) designs for network processors that employs traces to model the behavior of the key resources of the chip. The intent of this approach is to model the behavior of each resource abstractly while preserving the details of interactions between resources, with the goal of significantly increased simulation performance. Our mesoscale simulation also seeks to improve simulation performance, but with a more general approach. In particular, we propose detailed modeling for components of the system under investigation (SUI), with the remaining components modeled abstractly as the workload generators for the SUI. The approach of Wild et al. can be viewed as one instance of this general approach. Unfortunately, the authors do not give specific results on the improvement in simulation speed or on the accuracy of the resulting model.

Isshiki et al. [15] describe a novel encoding of a trace-driven workload as a *branch bitstream*, which simply records the sequence of outcomes (taken/not-taken) of each conditional branch in the application. The authors describe a straightforward approach to modifying the application source code to collect the branch bitstream. This information is then combined with the program's interprocedural control flow graph (ICFG) to drive the workload simulation. The authors use this infrastructure to simulate applications on a simple multiprocessor system-on-chip application and report speedups of up to 240x with almost no loss of accuracy. However, the authors concede that substantial extensions to this approach would be needed to properly model systems with complex memory subsystems and on-chip networks.

Schnarr and Larus [16] achieve a 5-12x simulation speed improvement over detailed microarchitectural simulation, while remaining as accurate as detailed simulation, by memoizing simulator actions for individual microarchitectural states and reusing the memoized actions upon encountering the same microarchitectural state at a future point in the program execution. If a particular microarchitectural state is not found in their processor action cache (p-action cache), their system falls back on detailed simulations to advance program execution and to memoize simulator actions for the newly discovered state. While their work applies memoization, a generic programming language technique, to simulator construction, our paper presents a methodology to achieve fast and accurate simulations for sub-system level studies. The two approaches are orthogonal and we believe memoization techniques can be used to further speed up mesoscale simulations.

Ipek et al. [17] train artificial neural networks with simulations of a small subset of a large space of simulator parameters and then use a machine learning technique called cross-validation to obtain performance predictions for the remaining (large) subset of the parameter space. The training continues until cross-validation yields sufficiently high accuracies. They are able to achieve a 98-99% performance prediction accuracy for the entire parameter space, by using only 1-2% of parameter combinations as training inputs. This falls under the realm of alternative strategies to conduct fast and accurate design space studies.

The Wisconsin Wind Tunnel simulator [18] executes target code directly on host hardware in order to simulate complex parallel systems efficiently. It abstracts processor pipeline models by annotating target executables with statically computed in-order pipeline cycle counts on a per basic block basis. This annotation enables direct execution to proceed unhindered while still obtaining realistic cycle estimates for the target code. While this work uses innovative simulator construction techniques, it still uses the traditional simulation methodology of executing an entire target program to obtain simulation results. Mesocale methodology differs in its use of well-defined abstraction

mechanisms to simulate in detail just the parts of a target application that matters to the SUI.

Finally we outline the relationship between mesocale modeling and queuing theory [19] [20] [21]. From the point of view of queuing theory, a mesoscale performance model is a special type of open queuing model with many sources posting a (potentially) infinite number of requests against the system under investigation. The interarrival times between the postings of requests are controlled by delays representing computation by the request source *as well as the response (feedback) from the SUI*. This latter feature seems to be rather atypical for modeling scenarios using open systems.

3 Mesoscale Performance Simulations

The key idea of mesoscale performance modeling is to combine methods from low- and high level performance modeling. In other words, we refrain from modeling the complete system and workload with the same level of detail but instead apply detailed modeling only to selected parts of the system and the workload. This results in simulations combining high accuracy with high efficiency and scalability for the subsystem and subset of the workload under investigation. Therefore, this methodology fills the gap between low- and high-level modeling. To explain this methodology in more detail, we first look at some of the key features of low- and high-level performance modeling.

In high-level modeling, the operations or requests of workloads are typically represented abstractly by probability distributions describing the interarrival and sojourn times at various components of the system [19] [20] [21]. It is important to note that workload semantics in the form of the sequence and dependencies between requests are typically not represented in this approach. While high-level modeling enables efficient and scalable performance simulations, it clearly lacks accuracy in cases where these features are important. For example, this is the case in detailed performance models of workload execution within processor cores where computation cannot proceed until data read from memory by a previous load request are available in registers.

High-level models of system components are typically implemented using either analytic queuing modeling [19] [20] [21] or discrete-event simulations [22] [23] [24]. Such models are represented by a network of delay and queuing centers without any detailed internal structure. Typical applications of this approach are in the domain of capacity planning of large IT installations or production sites or high-level design studies.

In contrast to high-level modeling, low-level models typically represent each request individually, in a form such as instruction traces or executables. This ensures a detailed representation of the workload semantics especially including the sequence of requests

and the dependencies between them. In this type of modeling, generally implemented in the form of discrete-event simulations [22] [23] [24], the system components are modeled in great detail. Typical applications of this approach are cycle-accurate processor simulations providing one of the key tools in contemporary processor design. A drawback of this approach is its extensive use of computational and human resources and its limited scalability.

The mesoscale performance modeling methodology is based on a hybrid strategy of decomposing the system into *Adaptive Workload Generators* (AWG) and the *System Under Investigation* (SUI). The workload is similarly decomposed into *delay requests*, which have no interaction with the SUI, and *key requests*, which involve processing by the SUI. This approach is illustrated in Figure 1.

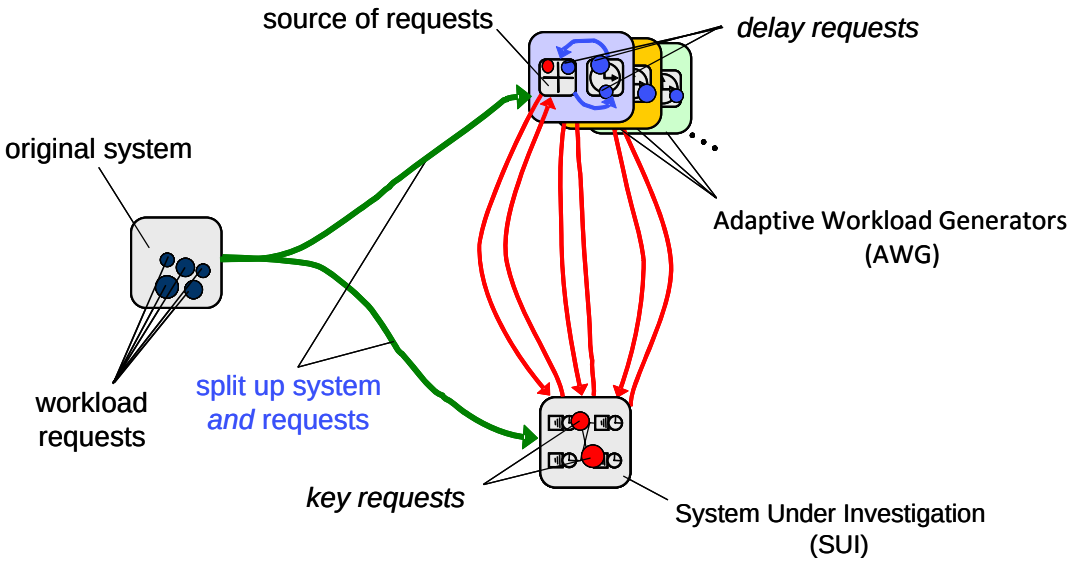

Figure 1: Decomposition of a system and a workload of a mesoscale performance model. The compound icon at the top represents the "Adaptive Workload Generator" (AWG) indicating its dual role as a source (represented by the "+" icon) of key requests posted against the SUI and of delay requests posted against its own delay center (represented by the "clock" icon). The icon for the SUI indicates an (arbitrarily complex) queuing network executing the key requests.

An AWG consists of a source for generating key requests and delay requests and a delay center. The source may be implemented using traces, executables or probability distributions. The AWG has the following tasks:

- posting key requests against the SUI
- posting delay requests against its own delay center
- controlling the rate of posting key requests by
  - executing the delay requests at its delay center between the posting of key requests
  - adapting the posting rates to the feedback from the SUI

The AWG and the delay requests represent the less relevant parts of the system and workload respectively (taking the modeling objectives into account) and are modeled on a high level.

The SUI and the key requests represent the parts of the system and workload respectively that are most relevant (taking the modeling objectives into account) and are modeled in detail. The SUI is responsible for executing the key requests posted by the AWG and providing appropriate feedback (response) to the AWG.

The AWG adapts the rate of execution of key requests based on feedback from the SUI. For example, the AWG may stall if the SUI can not accept further requests and resume posting of key requests when the SUI can accept new requests again.

Table 1 presents a summary of typical features of workload representation in low-level, high-level and mesoscale modeling methodologies.

Modeling methodology	Description of requests	Semantics
Low-level	each request is individually specified	For **all** requests respect: 1. sequence 2. dependencies 3. delays between posting 4. feedback from system executing the requests
Mesoscale	*key requests* are specified individually, others summarized as delays	Restrict 1.-4. to *key requests* only
High-level	requests specified using stochastic processes or average values	no concept of sequence or dependency

Table 1: Summary of the workload representation of mesoscale simulations versus typical workload representations of low-level and high-level modeling methodologies.

The mesoscale simulation methodology can be broadly applied, see Table 2 and Table 3 for some example scenarios. Although the modeling details will be different for each modeling scenario under consideration, we summarize some general principles to generate a mesoscale simulation:

- The first, and probably most important, step of a mesoscale modeling effort is to identify the parts of the system and workload mentioned above, namely the AWG, the SUI as well as the delay requests and key requests. Their identification is mainly driven by the modeling objectives, i.e. by the following questions:

  - What components of the system must be modeled in detail because their features or behavior may depend on the design alternative being evaluated, and therefore need to be part of the SUI?
  - What type of requests operate on these components and therefore need to be part of the set of key requests?
  - What components of the system have the role of generating these key requests and therefore need to be part of the AWGs?
  - What part of the workload controls the rate of posting of key requests at the AWG, i.e. can be subsumed in form of delay requests?

While there is no standard procedure for decomposing the system and workload in this fashion, the examples in Table 2 and Table 3 should provide some guidance on how to perform this activity.

- The second step is to specify these components and requests in detail:
  - The delay requests and key requests to be posted by the AWG can be specified in various ways, such as in the form of traces or probability distributions. The preferred approach for mesoscale modeling is to represent the sequence of requests as a trace which captures the essential workload semantics such as request ordering and dependencies. Often this trace can be produced from an execution of the workload on an existing system that is architecturally similar to the target system, or on a detailed full-system simulator. The key to this approach is to eliminate any effects of components of the SUI from the resulting trace, to allow the effects of alternative SUI models to be properly inserted during simulation. In Section 4 we describe how we have used this approach in modeling the Cell/B.E. interconnect and memory subsystem.
  - The delay center of the AWG is straightforward to model as a simple variable length delay. However, in some cases it may be appropriate to use a more sophisticated model to capture effects such as limited resources in the system components outside the SUI, such as a limited number of hardware threads of a processor core being used by several software threads.
  - The modeling of future SUI candidates as well the key requests will in general be based on the available information about the system architecture and the operation of the key requests on its components and interconnects. This information may be available in form of specifications or workbooks. The level of detail will in general be determined be the objectives of the modeling, the available resources and time constraints.
- Depending on the features of the scenarios to be modeled and the modeling objectives, it may be necessary to include the modeling of
  - backpressure -- the AWG is temporarily stalled by the SUI until resources are available or SUI processing has completed
  - synchronization -- the coordination of different requests
  - asynchronicity -- allowing key requests and delay requests to proceed concurrently

We describe how to implement all of these features in our Cell/B.E. modeling scenario below.

Complete system	AWG	SUI	Key requests	Delay requests
2-tier architecture	application server	Database server, disk, RAID, network	SQL queries	computational requests, memory I/O
hybrid multi-processor system	host processor	interconnect, accelerator	offload requests (e.g. XML encoding)	non-offloaded requests
processor	core	memory and bus subsystem, various controller	I/O requests against memory, network, disk	computational requests
search engine	user	search engine client and server, network	queries	thinking, drinking coffee, …
traditional IBM	IBM manager	IBM professional	work items	reading and writing mails, meetings

Table 2: A list of potential applications of mesoscale modeling for a simple setup consisting of just one AWG and a SUI.

Complete system	AWGs	SUI	Key requests	Delay requests
Cell/B.E.	SPE Cores	MFC, bus and memory subsystem	DMA operations	Computation by SPE cores
cluster	processor nodes	I/O subsystems, network	network I/O	computation by processor cores
hybrid multi-processor system	host processor cores	interconnect infrastructure, memory subsystems, accelerator cores	memory accesses, acceleration requests	computational by host processor cores
contemporary IBM	IBM manager of matrix organization	Many collaborating IBM professional teams	work items	reading and writing mails, meetings

Table 3: A list of potential applications of mesoscale modeling for more complex setups consisting of several AWGs and a SUI.

4 Mesoscale Simulation of the Cell/B.E. Processor

In this section, we apply the mesoscale performance modeling methodology to the Cell/B.E. processor [25], which is a multicore processor that supports asynchronous memory access. After providing a short introduction to the Cell/B.E., we discuss how to model the system and describe the decomposition into AWGs and the SUI. We then turn to a detailed discussion of workload characterization in the form of *distilled traces* and then discuss how to do workload generation based on these traces. Finally we present results of mesoscale simulation to various workloads on the Cell/B.E.

4.1 Overview of the Cell/B.E. Processor

The Cell/B.E. is a hybrid multi-core processor consisting of one general purpose core, the *Power Processing Element (*PPE), 8 special purpose cores called *Synergistic Processing Elements* (SPEs), two components for IO (IOIF0 and IOIF1) and the *Memory Interface Controller* (MIC) with the attached memory DIMMS. All these components are connected by the *Element Interconnect Bus* (EIB). Each SPE contains a single-instruction-multiple-data (SIMD) processing core called the *Synergistic Processing Unit* (SPU), a 256 KB *Local Store* (LS), and a *Memory Flow Controller* (MFC), which transfers data between the SPE Local Store and other elements of the memory subsystem. For further details on the Cell/B.E., please see [25].

The SPU cannot access main memory directly, but it can issue Direct Memory Access (DMA) commands to the MFC transfer data between local store and main memory. SPU computation and data transfers by the MFC can be performed *concurrently*, with synchronization of these activities controlled by the application. After issuing a DMA command to its MFC, the SPU may continue to issue subsequent DMA commands to the MFC, perform computations on data already available in its local store, or synchronize with previously issued DMA commands.

DMA commands that transfer data from main memory to the SPU's local store are called DMAGet commands, and DMA commands that transfer data from local store to main memory are called DMAPut commands. The SPU uses the mfc_get() and mfc_put() routines of the Cell/B.E. SDK 3.1 to issue DMAGet and DMAPut commands for a sequential region of storage. The MFC also supports list-form DMAGet and DMAPut commands, in which a sequence of Gets or Puts is specified by a list of address-data pairs residing in the SPU local store. List form DMAGets and DMAPuts are issued with the mfc_getl() and mfc_putl() routines of the Cell/B.E. SDK 3.1. A limit of 16 KB is placed on the amount of data that can be transferred by a sequential DMAGet or DMAPut

command or any list element of a list-form DMAGet or DMAPut. When performing data transfers, the MFC must break down, or *unroll*, each request into a series of memory subsystem transactions of at most 128 bytes each.

The EIB has separate command and data networks. Units that wish to perform data transfers on the bus first send a request to the bus on the command network. This request is received by the Address concentrator (AC) component of the EIB, which executes a coherence protocol for each request to ensure that the request obtains the most recent version of the data. When the coherence protocol is complete, the AC signals the appropriate bus units to perform the data transfer. Data transfer occurs on the data network of the EIB, which is organized as a set of four rings, with two running clockwise and two running counter-clockwise. Each ring is 16-bytes wide and is divided into 12 segments by the 12 units of the bus. The bus operates at half the speed of the processor and data is transferred in 16-byte chunks at a rate of one bus segment per clock cycle. The bus always uses the shortest path to send data between two units, which is at most 6 segments since the bus can send data in either direction on the rings.

Each DMA command has an associated tag, which is a value from 0 to 31. The SPU can check for completion of one or more DMA commands by issuing a synchronization request, called a *tag group status* request, to the MFC. This synchronization request contains a *tag mask*, which is a compact means to specify a set of tags. The SPU issues a tag group status request using a combination of the mfc_write_tag_mask() and mfc_read_tag_status_all() routines of the Cell/B.E. SDK 3.1. If all DMA commands issued by the SPU having one of the specified tags are complete, the MFC signals the SPU that the tag group is complete. Otherwise, the SPU blocks until the tag group completes. A DMAGet command is considered to be complete when the requested data have been delivered to the Local Store of the requesting SPE. This differs from DMAPut commands, which are considered complete when the associated data have left the SPE for the EIB bus.

A convenient way to understand these semantics is by looking at a few example sequence diagrams. Figure 2 presents a simplified sequence diagram for a workload executing on one SPU and accessing the memory subsystem (including the on-chip interconnect subsystem). After starting execution, the workload at the SPU performs some amount of computation. Computational instructions are represented as delays and indicated with red arrows in the time line associated with the core. Then it posts a data transfer request to the MFC. While the MFC is handling this request, the SPU continues executing further computational instructions, and then posts a second data transfer request to the MFC and so on. All these data transfers are handled by the MFC and underlying memory subsystem concurrently and asynchronously to the instructions executing at the SPU and

we can not make any assumptions about their timing behavior at the memory subsystem, e.g. they may even overtake each other.

The semantics of the synchronization requests may be understood as follows: when the workload at the SPU posts the synchronization request associated with data transfer request 3 posted earlier, the synchronization request stalls because the data transfer for request 3 has not yet completed. Some time later, the data transfer for request 4 completes, but the workload still can not continue executing at the SPU, because it is waiting for request 3 to complete. After some additional time, the data transfer for request 3 finally completes (e.g. the data to be retrieved from memory using a DMAGet have arrived in the local store) and the workload on the SPU can now continue execution. While doing so, the data transfer for request 1 completes. When the SPU reaches the synchronization for request 2 posted earlier, it stalls again because request 2 has not yet completed. Some time later, the data transfer for this request completes and the workload on the SPU can again resume execution. At the synchronization points for requests 4 and 1, there is no need for the workload on the SPU to stall, because the data transfers for these requests have already completed. Therefore, the workload on the SPU continues its computation until it finally reaches the end of its execution.

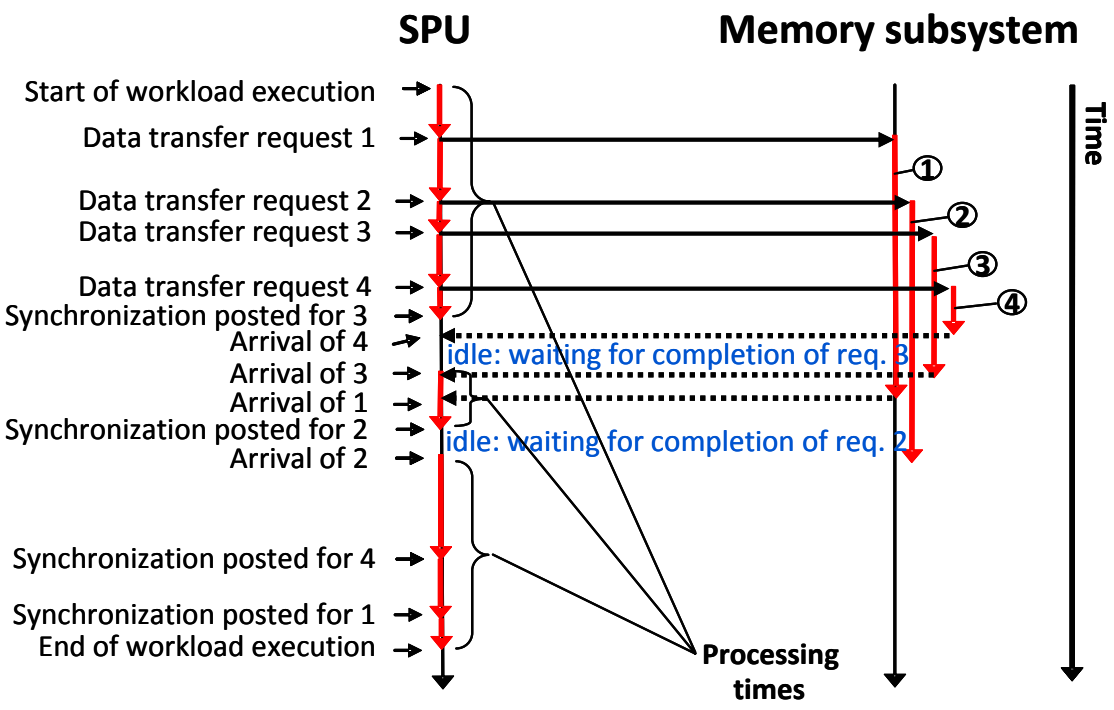

Figure 2: Sequence diagram for a workload executing on one SPU and performing asynchronous memory accesses through the MFC (including the on-chip interconnect subsystem).

It is also instructive to take a look at a typical source code snippet using asynchronous memory access operations, as shown in Figure 3. After some computations have been done, the workload posts an asynchronous load operation with tag 1. While this request is processed by the memory subsystem and on-chip interconnect, the workload proceeds with some computations and then posts another asynchronous load request with tag 2, followed by further computations.

Some point later in time, the workload may need the data requested by the asynchronous load request with tag 2; therefore it posts a synchronization request to indicate that the data for the load request must be available for the workload to continue execution. If this is already the case, the workload may immediately continue; otherwise the workload stalls until the data for the asynchronous load request is available. Later in the code, a similar synchronization request is issued for the data associated with the asynchronous load request 1.

```
// Computations...
...
// Post asynchronous load operation with tag 1
asyncLoad(1,...);
// Continue with computation while load is processed concurrently
...
// Post asynchronous load operation with tag 2
asyncLoad(2,...);
// Continue with computation while load(s) is (are) processed concurrently
...
// Synchronize for the second load operation
sync4Load(2,...);
// Continue with computations using the data of the second load operation
...
// Synchronize for the first load operation
sync4Load(1,...);
// Continue with computations using the data of the first load operation
...
```

Figure 3: A code snippet of a workload using asynchronous memory access and associated synchronization operations.

4.2 Modeling Objectives

The objective of our modeling effort for the Cell/B.E. is to determine the performance impacts of varying memory subsystem and on-chip interconnect designs. In particular, our goal is to determine the performance characteristics of a data cache located in each SPE to cache data transferred between the SPE and main memory. The Cell Broadband Engine Architecture [26] specifies that such a cache may be present, but the Cell/B.E. processor does not implement this optional architectural feature. With this objective in mind, our primary focus for the modeling effort is the memory subsystem.

To understand how this modeling objective affects our approach to a mesoscale model for the Cell/B.E., we consider how a faster memory subsystem can affect the performance of Cell/B.E. workloads. Figure 4 presents a possible sequence diagram for the same workload illustrated in Figure 2, and assuming the same SPU core and delays, but with a new and faster memory subsystem. In this case, the requests accessing the memory subsystem have *always* completed when the workload on the SPU requires their completion in order to continue execution. Therefore, the waiting times have disappeared completely and the overall processing time is significantly shorter. Obviously, introducing an even faster memory subsystem (or on-chip interconnect) would not result in a further reduction of processing time for this workload.

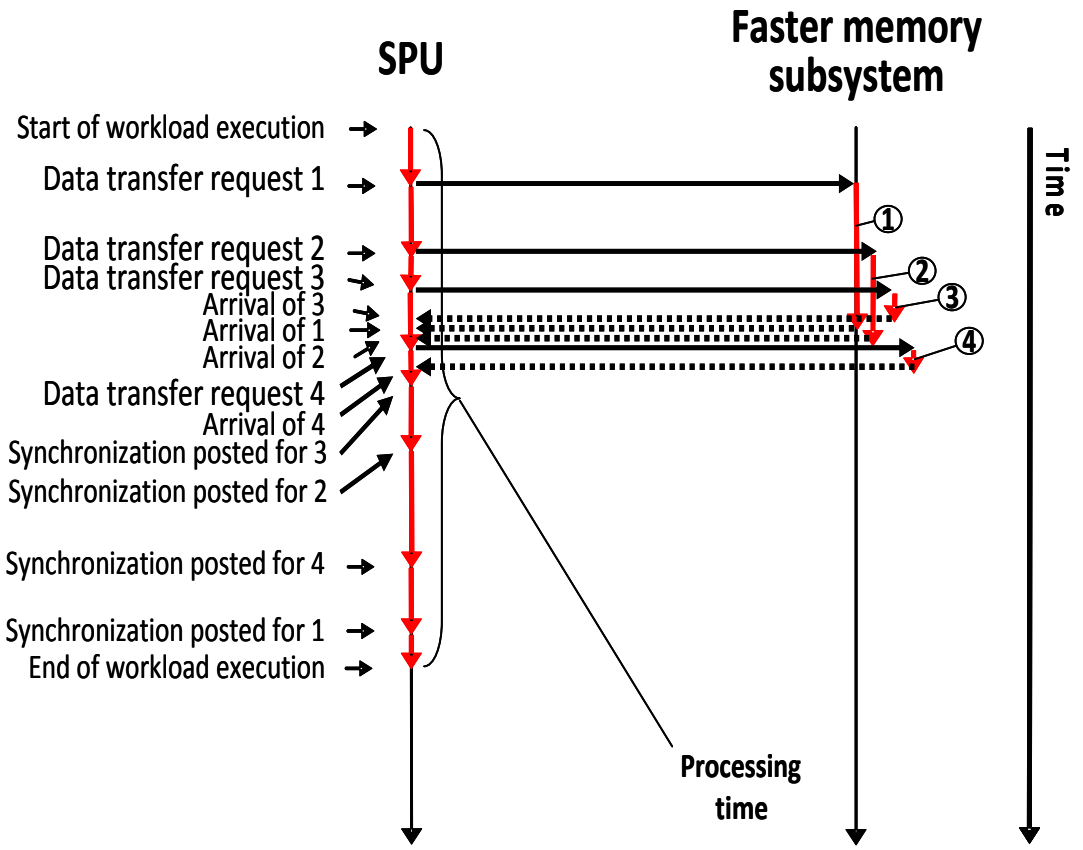

Figure 4: Sequence diagram for a workload executing on one SPU and accessing the memory subsystem (including the on-chip interconnect subsystem) using a faster memory subsystem than in Figure 2.

4.3 System and Workload Decompostion

As described above, the system must be decomposed into the AWGs and the SUI. Because our modeling objective is a design space evaluation of the memory subsystem of the Cell/B.E. processor, the memory subsystem, consisting of the on-chip interconnect (the EIB), the MIC and the memory DIMMS must certainly be a part of the SUI. The MFC plays a significant role in scheduling DMA operations to the EIB, and all DMA transfers involve either reads or writes to Local Store, so both of these components are also included in the SUI. As a result, the border between the AWGs and SUI is located *inside* the SPEs. Figure 5 illustrates our decomposition of the Cell/B.E. system into AWG and SUI.

The decomposition of the workload is rather straightforward. As key requests we identify the requests accessing the memory subsystem, which can be classified as DMA requests (i.e. DMAGets and DMAPuts) or synchronization requests. All other computation performed at the SPU cores is represented as delay requests. The sequence of requests in the workload is captured in a *distilled trace*. The distilled trace is s attached to each AWG is described below.

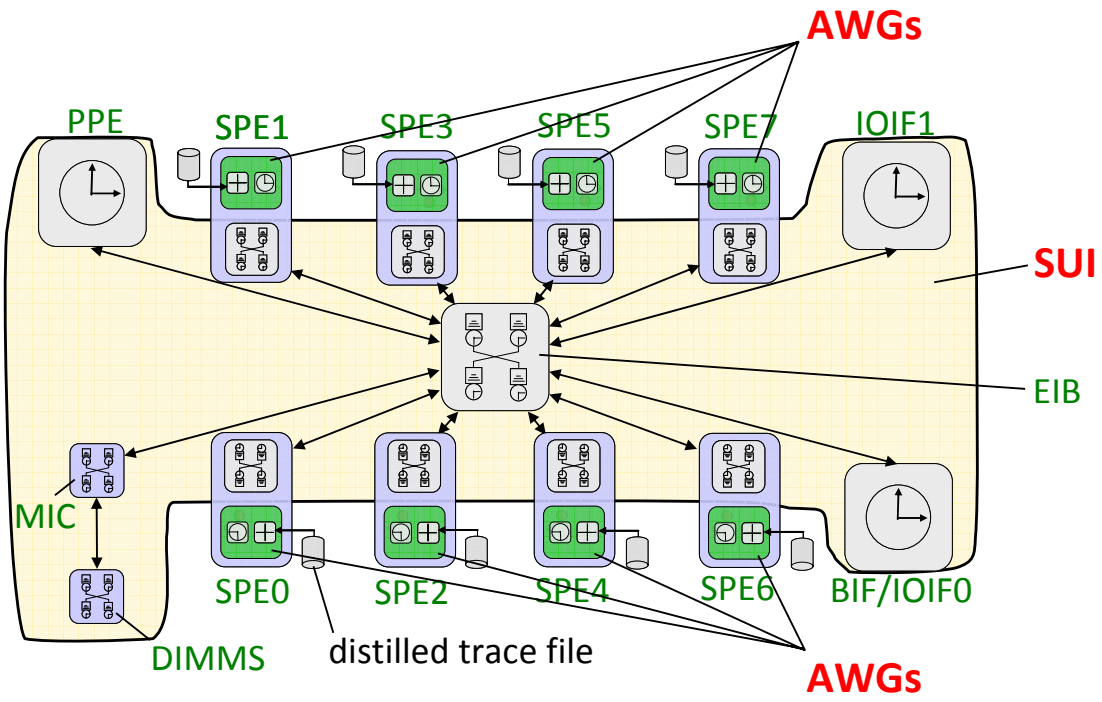

Figure 5: Modeling setup for our mesoscale performance model of the Cell/B.E. processor indicating the AWGs and the SUI consisting of the EIB interconnect and the memory subsystem. Attached to the source of each AWG is a file containing the associated distilled traces.

Because our current focus are workloads executing on the SPEs and accessing the system memory using DMA operations, the PPE and the IOIFs have not been implemented in detail.

4.4 Workload Characterization

The workload characterization of our model consists of patterns of *computational*, *DMA* and *synchronization requests* (see Figure 6) with:

- *Computational requests* representing computation by the SPU cores, which are processed by the delay center of the AWG,
- *DMA requests* representing DMA commands issued by the SPU, which are processed by the MFC, Local Store, EIB, MIC and the Memory DIMM components of the SUI,
- *Synchronization requests* representing operations by the SPU to ensure completion of a set of DMA requests, which are processed by the MFC component of the SUI.

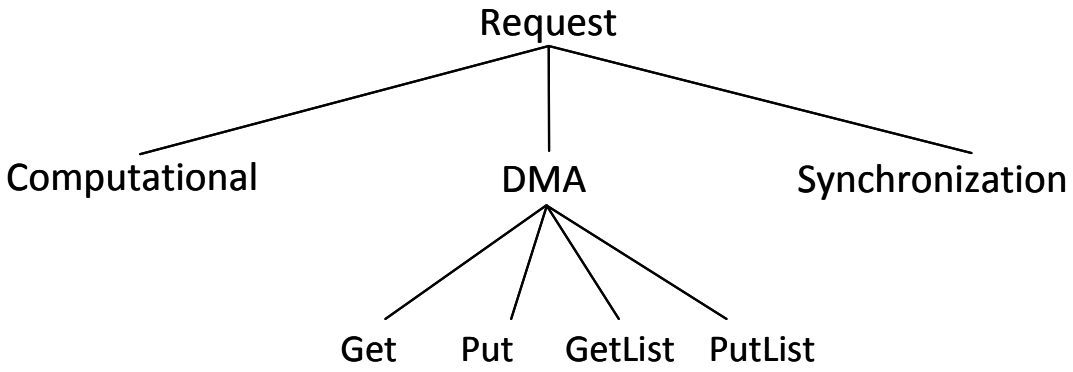

Figure 6: The hierarchy of requests included in our model.

The set of DMA requests modeled consists of DMAGet and DMAPut requests and the associated list versions, DMAListGet and DMAListPut. We also model barrier and fence versions of these requests. A synchronization request blocks the AWG until all previously issued DMA operations associated with this synchronization request have completed. For more details, see [27].

Next we provide an overview of how to obtain the workload characterization (i.e. the patterns of requests) that is used in the adaptive workload generation.

Figure 7 presents an overview of the process to create the workload characterization. We start with instruction traces generated by a current Cell/B.E. system or the associated IBM full-system simulator. From traces, we generate distilled traces containing only the

computational requests, DMA requests and synchronization requests as described in more detail in the next paragraph. These distilled traces are then used by Adaptive Workload Generators to create the stream of requests simulating the workload of a mesoscale simulation. The latter step is outlined in the next section.

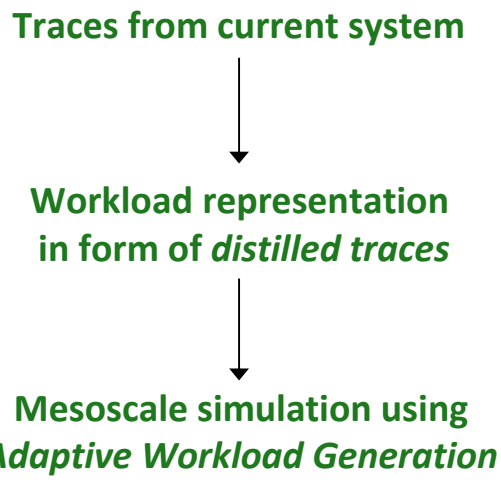

Figure 7: Overview of the process to create and use the workload characterization for the mesoscale simulation of the Cell/B.E.

The input to the trace distillation process is an instruction trace for each SPE captured during the execution of the workload on a current Cell/B.E. system or the IBM Full System Simulator (SystemSim) for the Cell/B.E [28]. The trace includes a timestamp for each instruction indicating the cycle at which the instruction was issued. This can be done using tracing mechanisms on actual hardware or with tracing facilities of a cycle-accurate simulator. While the first option could be much faster, we chose the latter option because the flexible tracing and triggering facilities of SystemSim made it easy to design a trace collection process that captured all the relevant information. Since we only need to execute the workload once to capture the trace of workload requests – not for each design alternative to be explored – this approach turned out to be quite feasible for moderately sized workloads.

A distilled trace contains:

- computational requests to be executed by the SPU core in form of delays
- DMA requests including their
  - target addresses

    - data sizes
    - number of list elements (for list requests only)
    - tag
    - barrier flags
- Synchronization requests including their
    - tag mask

It is important to note that the sequence and dependencies of the DMA and synchronization requests are preserved in the distilled traces, but all time stamps associated with instructions of the original trace files have been eliminated.

The key steps of generating the distilled traces using the IBM SystemSim are as follows:

1. Scan the instruction trace of each SPE for instructions that identify the key requests of the SPE. DMA requests are identified by write channel (wrch) instructions to channel 21, and synchronization requests are identified by a read channel (rdch) instruction for channel 24.
2. For each key request identified in step 1, collect the attributes of the request. The attributes for our key requests are all values of MFC channels whose values can be determined by scanning backward in the trace to the last point where the channel was written.
3. Determine the amount of time spent in computation between each successive pair of key requests. This time is the difference between the timestamp of the instruction *following* the wrch/rdch of the first key request and the wrch/rdch of the subsequent key request. The time to perform the wrch/rdch instruction is dependent on the implementation of the SUI, and thus is excluded from the distilled trace. This computation is represented in the distilled trace with a computation request for this amount of time.
4. Now all key requests and delay requests have been characterized. For each SPE, write the sequence of requests to the distilled trace file, respecting the order in which they occur in the original instruction trace.

4.5 Workload Generation

The AWGs utilize these distilled traces to generate the workload for the simulation as follows:

- The AWG for each SPE reads entries from its distilled trace

- Computational requests (i.e. delay requests) are modeled simply as delays by the AWG as a means to control the posting rate of key requests
- DMA or synchronization requests (i.e. key requests) are posted by the AWG against the SUI respecting their sequence

Figure 8 illustrates the operation of the AWG. The adaptation of the workload generation is accomplished by adapting the rate at which key requests are posted according to the sojourn time of the computational requests and the backpressure from the SUI in accepting key requests. The SUI creates backpressure by stalling the AWG when it makes a key request that must wait for SUI resources or processing to complete.

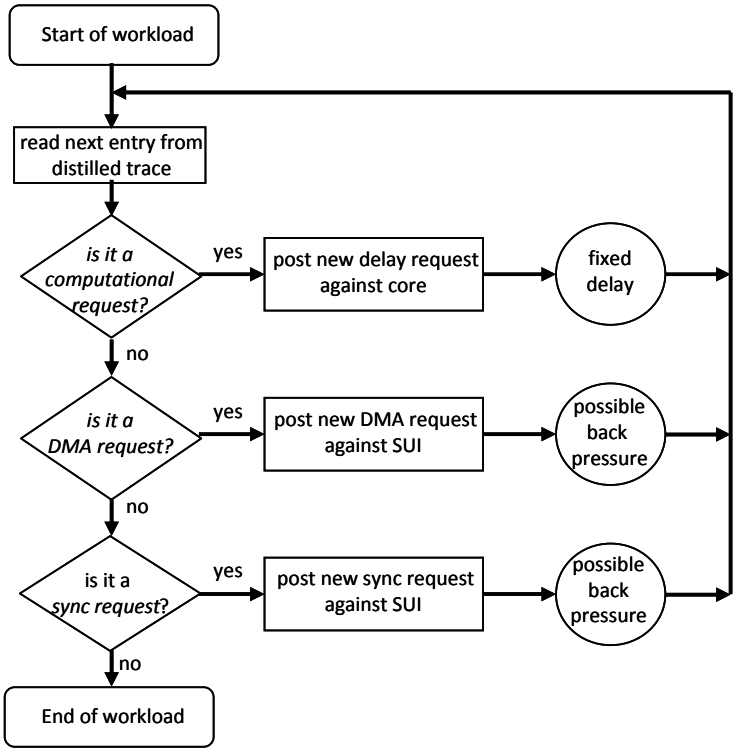

Figure 8: Operation of the AWG. Backpressure from the SUI on key requests adapts the rate of workload generation based on system design.

In the case of a computational request, the AWG delays for a fixed time interval specified in the request and then proceeds to read the next entry from the distilled trace. In the case of a *DMA request*, the request is posted to the SUI. The SUI could create backpressure

by stalling the AWG when this request is posted. For example, a stall occurs when a DMA request is posted when there is no available slot in the MFC command queue for the new DMA request. As soon as a slot becomes available in the MFC command queue, the SUI returns to the AWG which then continues with the next request in the distilled trace. In the case of a synchronization request, the request is posted to the SUI, which stalls AWG processing until the data transfers for the specified DMA requests have completed. Because the time to complete data transfers depends on the performance of the SUI, back pressure from synchronization requests impacts the posting rate of new delay requests and therefore also adapts the posting rate of new key requests to the performance of the SUI.

4.6 System Modeling

Our approach to modeling the SUI for the Cell/B.E. is based on standard discrete-event simulation methods [22], [23], [24]. The main advantage of these methods is that they allow detailed cycle-accurate modeling to be combined with high-level queuing models. This feature nicely supports the mesoscale modeling approach.

Each component of the SUI is represented by a module. Each module may have ports to allow requests entering and leaving the module. These ports are then connected via links to form a network of modules representing the complete system consisting of connected components. Requests entering the SUI from the AWG could be initially routed to any component of the SUI, but in our model are typically routed first to the MFC. Components process a request by applying a service time and then generating one or more requests on its outbound links to modules representing other components or back to the AWG, as specified by the system specifications. The arrival and departure of requests at the modules are the events of the model that are tracked in the dynamically adapted Future Event Set, the key data structure of any discrete-event simulation. The event handlers associated with these events implement the details of the operations of the requests at the modules.

The component models are constructed in a hierarchical manner, which allows each component to be further decomposed into a network of models that perform various functions of the component. For example, the SPE component is decomposed into models for the SPU, Local Store, and MFC, and the EIB component is decomposed into models for the Address Concentrator (AC) and the data transfer rings. The modeled components of the Cell/B.E. and the memory subsystem are shown in Figure 9. They are specified by various attributes, such as:

- latencies (sojourn or service times) for all types of requests operating at that component

- maximum number of requests in flight allowed at that component to model (finite) resources

The operations of modeled request types at the components of the model are shown in the component-request matrix of Table 4.

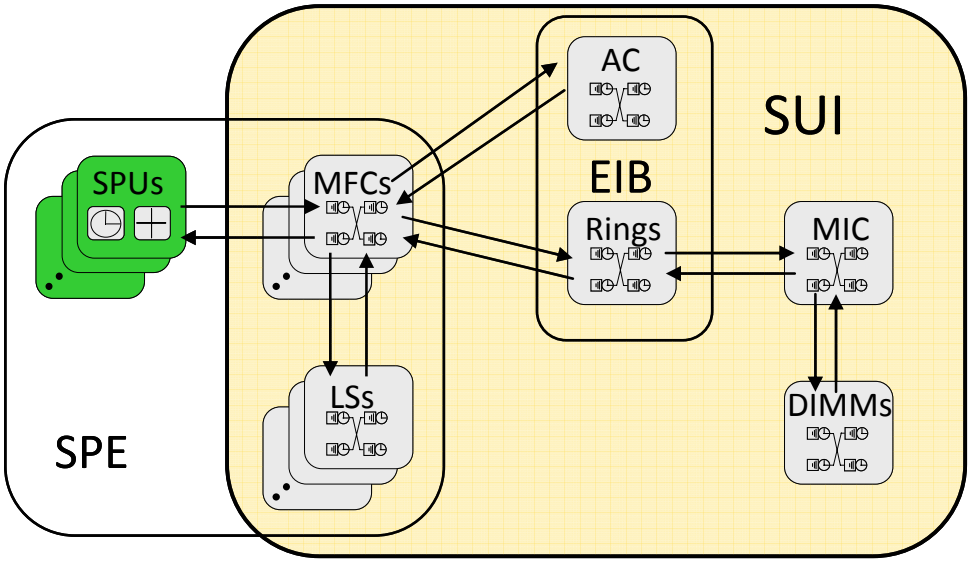

Figure 9: Network of system component models.

In our mesoscale modeling of the Cell/B.E system, we have made several approximations:

- We assume that the latencies are independent of the internal state of the component.
- We assume that a component may temporarily block entry to new requests arriving at the component, but once admitted, all requests are processed with the same sojourn time. All components use a simple first-come-first-served arbitration policy to allocate shared resources to requests. Requests that are blocked may result in a backpressure chain that blocks new key requests from entering the SUI from the AWG.

	SPU	MFC	Local Store	EIB AC	EIB Rings	MIC	Memory DIMMs
Computational	delay	—	—	—	—	—	—
Synchronization	—	sync. for DMA requests	—	—	—	—	—
DMAGet	—	create and rejoin unrolled reqs	—	—	—	—	—
DMAGetUnrolled	—	created, rejoined	read, write	coherence protocol	send data from src to dst	memory access	read from ranks
DMAPut	—	create and rejoin unrolled reqs	—	—	—	—	—
DMAPutUnrolled	—	created, rejoined	read, write	coherence protocol	send data from src to dst	memory access	write to ranks
DMAListGet	—	unroll and rejoin all reqs of list					
DMAListPut	—	unroll and rejoin all reqs of list					

Table 4: The component – request matrix shows the operations executed by the various requests at the components of the model.

- We did not model contention between SPU loads and stores accessing the Local Store concurrently with the DMA requests. This would have required us to explicitly model the load and store requests of the SPUs as key requests, which would have significantly reduced the benefit of our approach. Instead, we performed a sensitivity analysis by introducing background requests at the Local Store with varied intensity. This revealed that the impact of the contention for the Local Store is not significant for the workloads under consideration.
- We did not model details of the coherence protocol, but included appropriate delays and serialization points for the DMA requests with the command and data paths of the EIB.
- We introduced a simple probabilistic arbitration scheme for the data paths of the EIB. When more than one data path is available (maximal two) with the same length and destination, one of them is selected with equal probability.
- We did not model the *translation lookaside buffer* (TLB) of the MFC. All the workloads we chose to evaluate use huge pages, which significantly reduce the impact of address translation on application performance.

As an example for modeling a DMA operation, Figure 10 presents the execution path of a DMAGet (load) request posted by SPE1. After reading the DMAGet request from the

distilled trace file, the AWG posts the request to the MFC, which is part of the SUI. In this case, there is a slot available in the MFC command queue, so the SUI exerts no back-pressure on the AWG – it accepts the DMAGet request and returns to the AWG. The AWG reads the next record from the distilled trace and finds that it is a computational request. This request is posted to the delay center of the AWG and these two requests are now processed concurrently by the SUI and AWG. The DMAGet request is unrolled at the MFC into 128 Byte requests which are then routed to the MIC via the EIB command network. The unrolled requests are indicated by the dashed lines in Figure 10. After retrieving the data from the memory DIMMS, the MIC routes the data via the EIB bus back to SPE1, the source of the request. At SPE1, each unrolled request delivers its data to the Local Store. Once all of the unrolled requests are done, the original DMA request is marked complete.

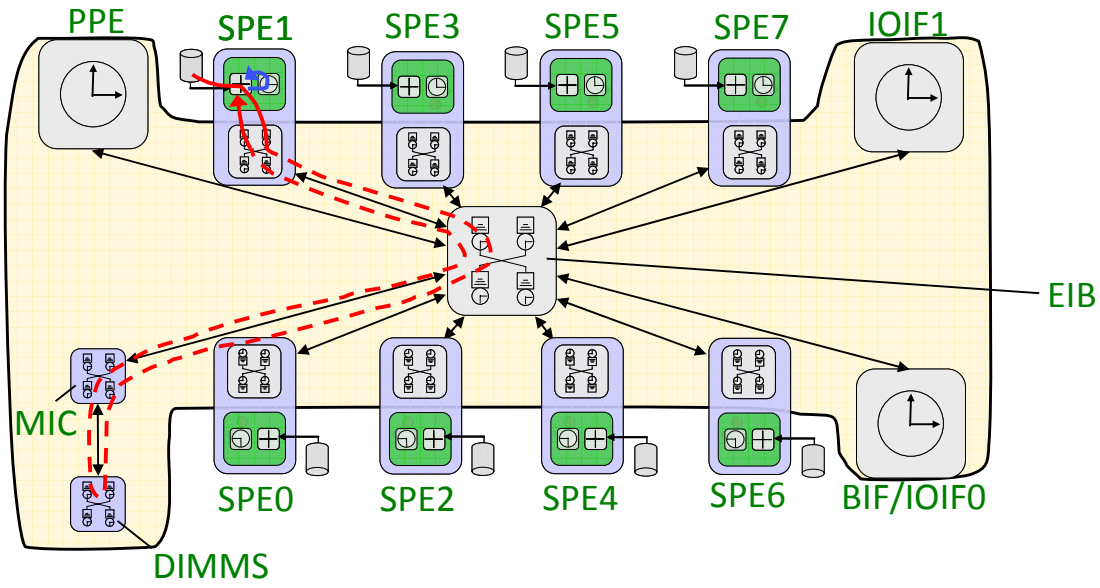

Figure 10: The execution path of a DMAGet operation posted by SPE1 that accesses the memory DIMMS via the EIB and the MIC.

Figure 11 illustrates the decomposition of the components of the SPE between the AWG and SUI. As discussed above, the border between the AWG and the SUI divides the SPEs in two parts: the SPU cores are part of the AWG while the MFC and the Local Store are part of the SUI.

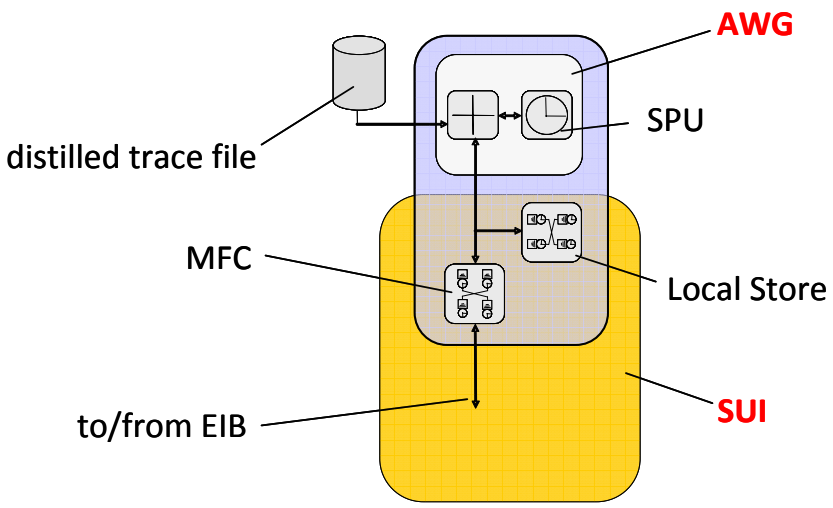

Figure 11: Decomposition of the SPE between the AWG and the SUI.

The MFC is a key component of our simulation because it implements key details of DMA and synchronization requests. The MFC must handle incoming DMA and synchronization requests from the AWG and interacts with the memory subsystem to perform the requested data transfers. The interface from the MFC to the memory subsystem is asynchronous, meaning that the MFC submits requests for data transfer to the memory subsystem, and at some later time the memory subsystem signals the MFC that the transfer is complete. The MFC tracks all outstanding DMA requests and data transfers so that it can report completion of these requests to the AWG in response to synchronization requests.

Figure 12 illustrates the processing performed by the MFC in our mesoscale simulation. The left side of the figure illustrates the handling of key requests coming from the AWG. In the case of a DMA request, the MFC allocates a slot at the command queue of the MFC to hold the DMA command. If no slot is available, the request blocks within the MFC, creating back pressure to the AWG. When a slot is available, the MFC allocates the slot, stores the DMA request, and returns to the AWG. In the case of a synchronization request, the MFC checks whether the specified DMA requests have all completed. If so, the MFC simply returns to the AWG. Otherwise, the MFC blocks, again creating back pressure on the AWG, until all the specified DMA requests have completed. Recall that a DMAGet is complete when the requested data has been delivered to Local Store, while a DMAPut is complete when the associated data has been delivered to the EIB.

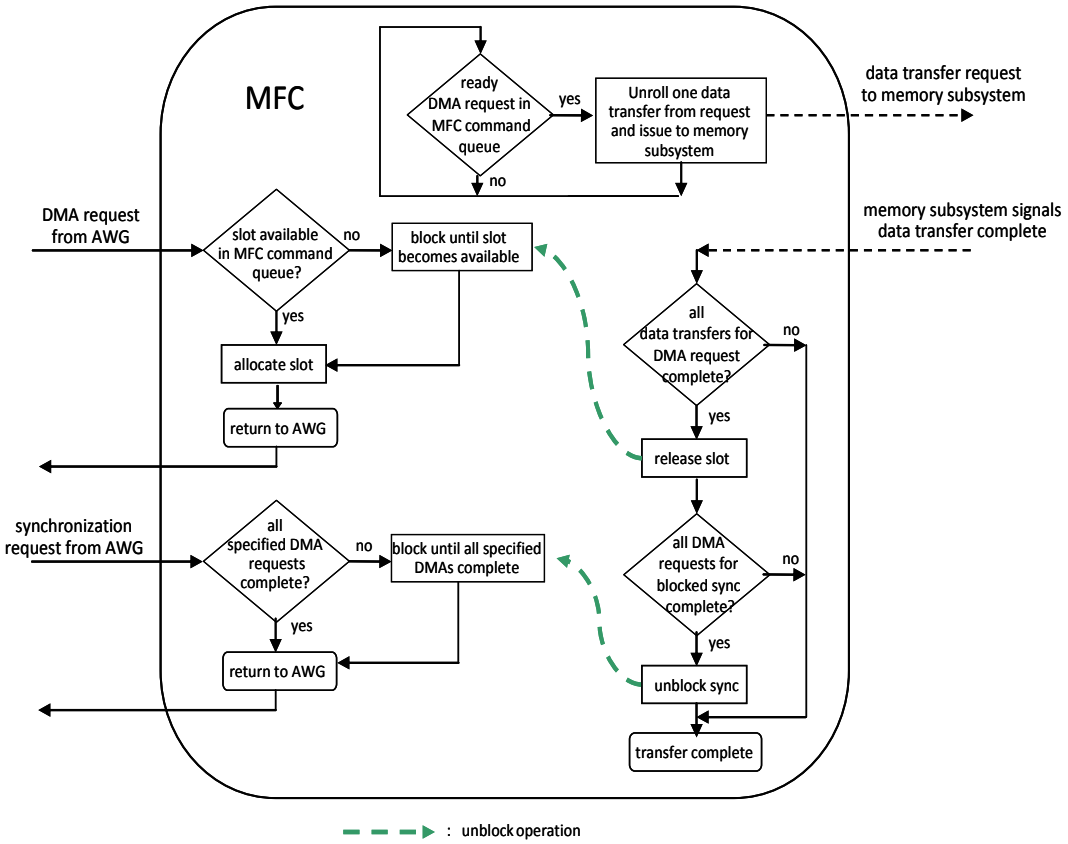

Figure 12: Processing performed by the MFC in our mesoscale simulation.

A separate thread of control in the MFC, the *unroll thread*, handles the generation of data transfer requests to the memory subsystem. The unroll thread breaks each DMA request into a sequence of data transfer requests to the memory subsystem, each with a maximum size of 128 bytes. The unroll thread checks the MFC command queue looking for a DMA request that requires additional data transfers that are not already in flight. The unroll thread submits data transfer requests to the memory subsystem at regular intervals determined by the system design parameters.

The right side of Figure 12 illustrates the MFC processing for a signal from the memory subsystem indicating a data transfer has completed. A data transfer for a DMAGet is considered complete when its data has been delivered the Local Store, and a DMAPut is considered complete when its data has been retrieved from the Local Store and delivered it to the EIB. When the last data transfer for a DMA request completes, the slot in the MFC command queue for this request can be released. This could allow a previously

blocked DMA request to unblock and obtain this slot to store the new DMA request. If a previously issued synchronization request was blocked waiting for this DMA request to complete, and all the other DMA requests for this synchronization request have completed, the synchronization request can unblock and complete processing by returning to the AWG.

5 Evaluation

5.1 Methodology

To evaluate our mesoscale modelling methodology, we have used 2 different classes of benchmarks to compare the results or our mesoscale simulations against a real hardware system and a cycle-accurate full-system simulation model, the IBM Full System Simulator for the Cell/B.E. processor (SystemSim) [28]. Our reference hardware platform is an IBM BladeCenter QS21 that has been modified to use only one Cell Processor. The mesoscale and SystemSim simulations were configured to model this reference hardware platform.

The first class of benchmarks consists of basic DMA commands – DMAGets, DMAPuts, and list-form DMAGets and DMAPuts. For sequential DMAGets/DMAPuts, we vary the data size from 8 B to 16 KB, and for the list-forms of DMAGet/DMAPut we vary the data size from 128 B to 128 KB. The DMA commands are initiated from a single SPE while all other SPEs are idle. To increase precision, the benchmark reports the average latency of 100 executions of the subject DMA command performed in succession.

The second class of benchmarks consists of application kernels, where we have chosen the FFT16M of the Cell SDK Version 3.1 as a representative example. This benchmark performs a one dimensional FFT on 16 million single precision complex data values. The benchmark can be executed on 1, 2, 4, or 8 SPEs.

5.2 Accuracy

To assess the accuracy of the performance results of our mesoscale simulation model, we compare the actual execution time measured on the IBM BladeCenter QS21 with the simulated execution times from our mesoscale simulation and from the IBM Full System Simulator for the Cell/B.E. for each of our selected benchmarks. Given the rather high level of modelling abstraction of our approach, our accuracy target was to keep the deviations of our mesoscale simulation results versus hardware measurements below 10%. In fact in most cases this goal has been overachieved.

Figure 13 shows results for the DMAGet benchmark when the target of the DMAGet is a location in system memory. The x-axis of the graph indicates the data size of the

DMAGet command, from 8 B to 16 KB, and the y-axis indicates the time in processor clock cycles (pclocks) needed to perform the data transfer. Note that all data transfers on the EIB are performed in blocks of 128 bytes, so the latencies for all requests with data sizes of 128 bytes or less are identical. For this benchmark, results from our mesoscale simulation are within 8% of the actual hardware performance for all data sizes, and even exceed the accuracy of the results of the low-level SystemSim model for data sizes of 1024 bytes and larger. It is somewhat surprising that the mesoscale simulation can achieve greater accuracy than the low-level SystemSim model. However, SystemSim is used to model a wide variety of PowerPC architecture systems, so its developers must sometimes trade accuracy of the results on a particular system configuration against the impacts on results of other configurations.

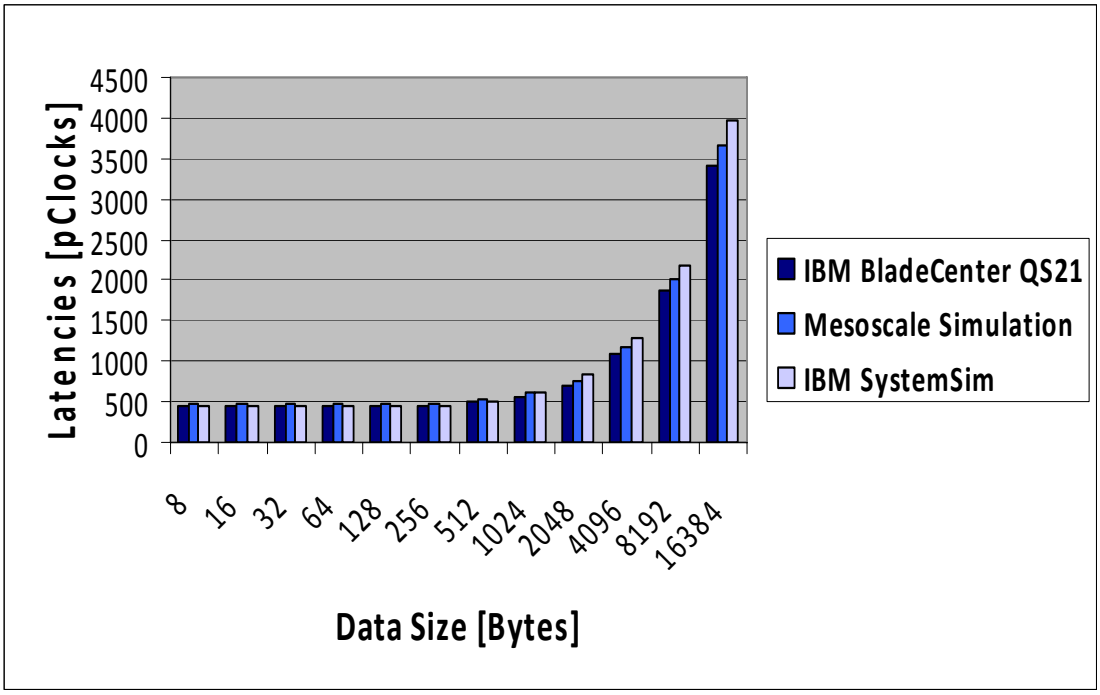

Figure 13: Simulation results and measurements of DMAGet latencies for reading data from system memory for varying data sizes.

Figure 14 shows the analogous results for posting DMAGet requests against the Local Store of another SPE. The latencies for accesses to Local Store of another SPE are significantly below the ones for accessing system memory mainly because the there is no off-chip communication required to obtain the requested data. For this benchmark, the performance predicted by the mesoscale model is within 9% of the performance on

hardware, and within 1% of hardware for data sizes of 2048 and larger, which is roughly equivalent to the accuracy achieved by the low-level SystemSim model.

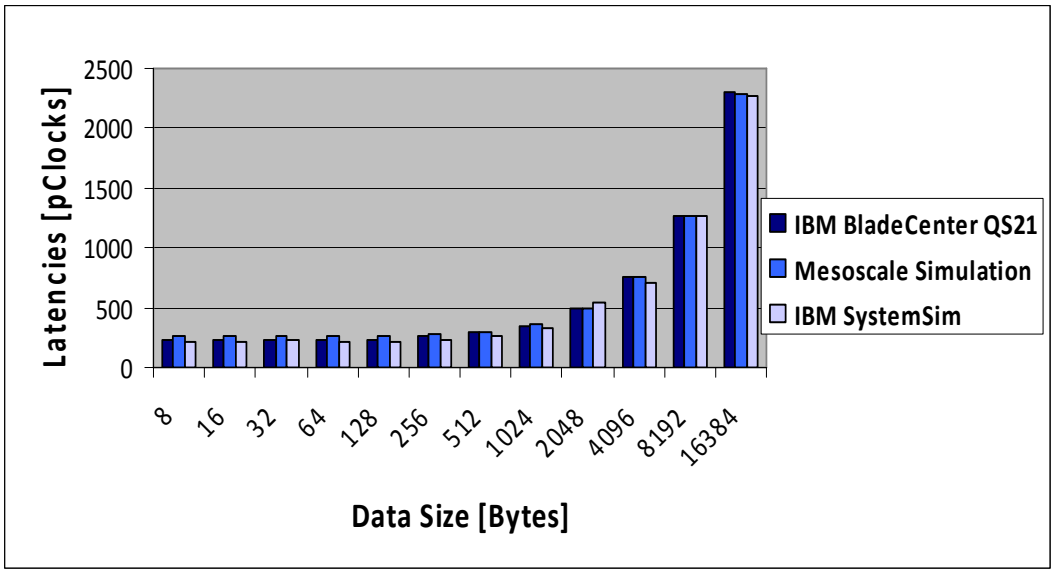

Figure 14: Simulation results and measurements of DMAGet latencies for reading data from the Local Store of another SPE.

Figure 15 shows results for our DMA benchmark posting DMAPut requests against the system memory. Recall that a DMAPut is considered complete when the associated data have been delivered to the EIB. This differs from DMAGet commands, which are not considered complete until the requested data have been delivered to the Local Store. This means that the latency of a DMAPut operation, as perceived by the SPE, excludes the time needed to actually store the data to the memory DIMMs. This explains the lower latencies of DMAPuts versus DMAGets. Except for a data size of 8 bytes, the deviation between the mesoscale simulation results and the hardware measurements are less than 7%, which exceeds our 10% accuracy goal.

The large deviation of the mesoscale simulation result and the hardware measurement for 8 byte DMAPuts is caused by the special processing performed by the memory controller to update the error-correcting codes (ECC) for stores smaller than 16 bytes (the ECC granule size). This increases the occupancy of the Memory DIMMs to the point that this component becomes the bottleneck for a series of 8 byte DMAPut operations. We have not yet implemented this behavior in the mesoscale simulation.

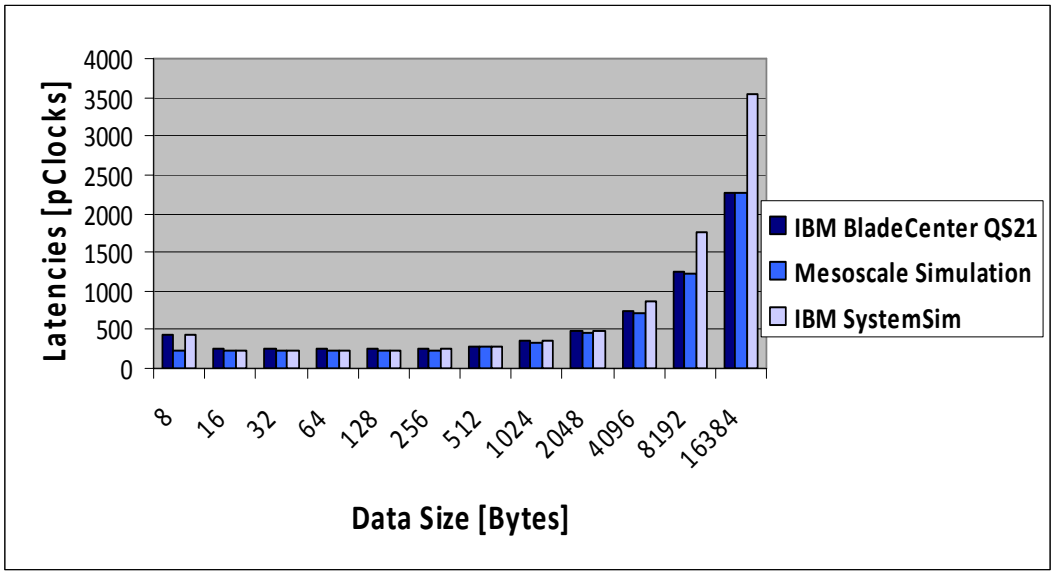

Figure 15: Simulation results and measurements of DMAPut latencies for writing data to system memory.

Figure 16 shows the analogous results for posting DMAPut requests against the Local Store of another SPE. For this benchmark, the performance predicted by the mesoscale model is within 5% of the performance on hardware for all data sizes, which is roughly equivalent to the accuracy achieved by the low-level SystemSim model. Here the anomaly for 8 byte transfers does not occur. This is because the Read-Modify-Write operation is integrated into the SPE Local Store access path, and thus does not create a bottleneck even at very high access rates.

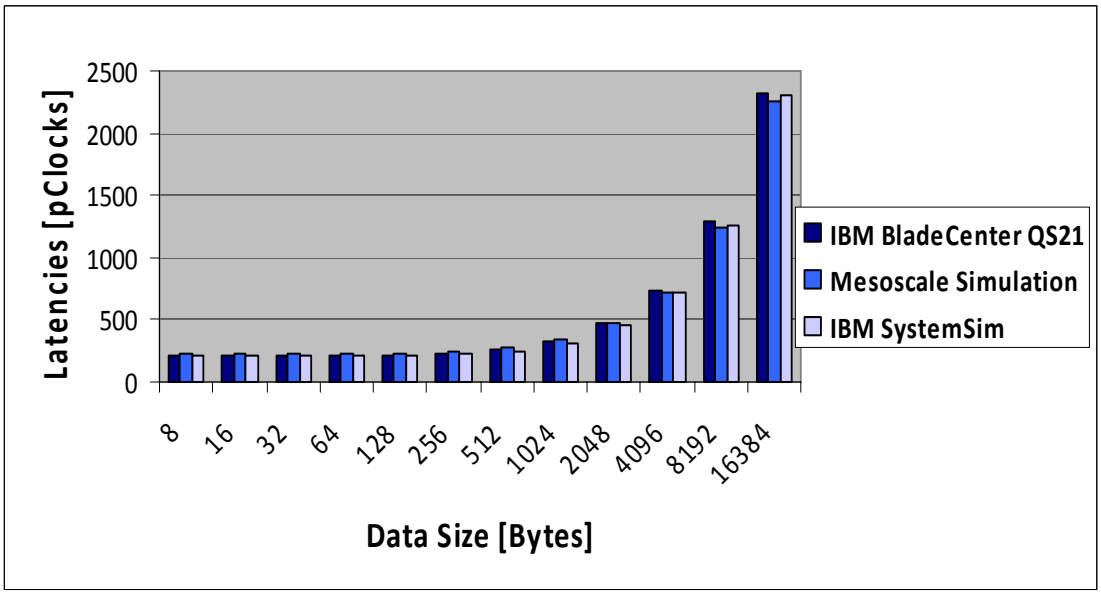

Figure 16: Simulation results and measurements of DMAPut latencies for writing data to the Local Store of another SPE.

Figure 17 and Figure 18 present the latencies of DMAGetlist and DMAPutlist commands for data in system memory. In our benchmark, each list entry specifies a transfer size of 128 bytes, with the number of list entries varying from 1 to 1024. Because these list operations require additional accesses of the Local Store, the latencies are significantly higher compared with ones of the simple DMAGet and DMAPut requests. For both the DMAGetlist and DMAPutlist operations, the mesoscale simulation results are within 6% of hardware performance for all data sizes, and within 2% for data sizes of 4096 bytes and larger.

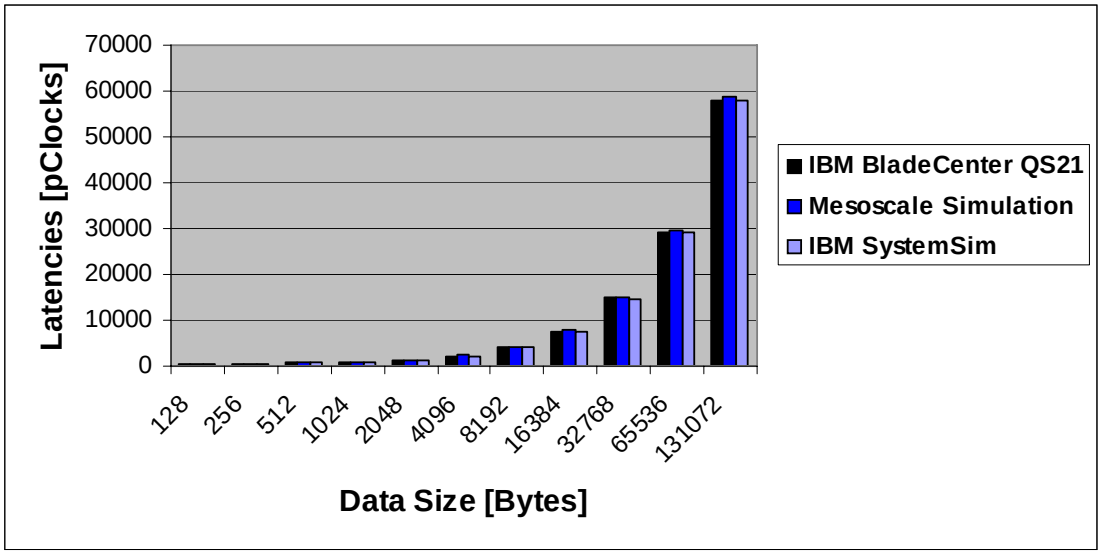

Figure 17: Simulation results and measurements of DMAGetlist latencies for reading lists of data to memory.

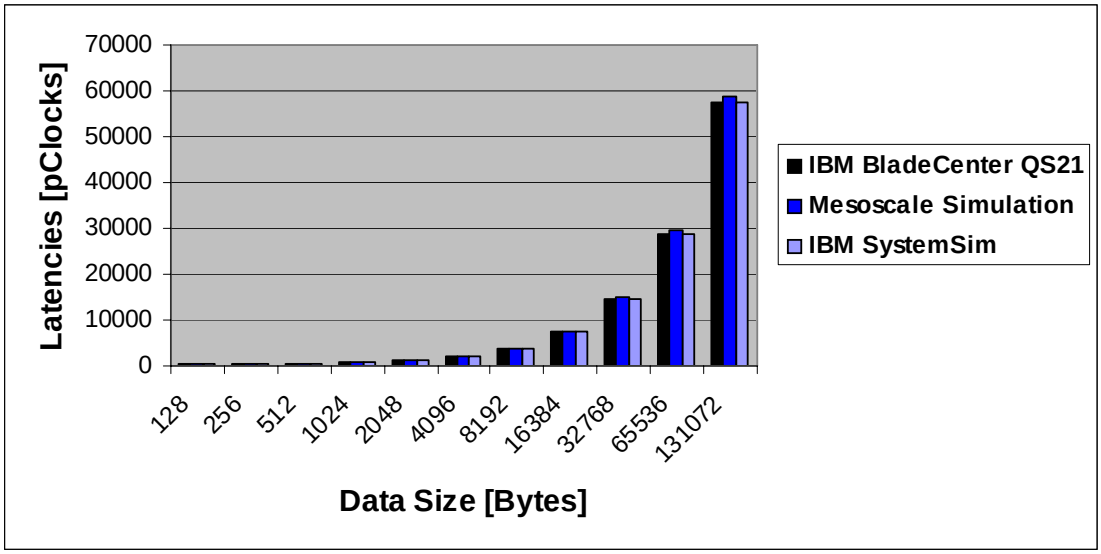

Figure 18: Simulation results and measurements of DMAPutlist latencies for writing lists of data to system memory.

Finally, Figure 19 presents the execution times for a more complex workload compared with the ones above, namely the FFT16M of the SDK3.1, for various numbers of SPEs. This workload performs a rather complex pattern of DMAGetlist and DMAPutlist requests including associated synchronizations. Because FFT16M overlaps most of its communication and memory accesses with computation, the execution time decreases almost linearly as a function of the number of SPEs. Accuracy of the mesoscale simulation model is excellent for both 1 and 2 SPEs executions, with less than 1% error. Results for 4 SPEs are still very good, where mesoscale simulation produces results within 3% of the measured hardware performance. For 8 SPEs, the mesoscale simulation overestimates performance by approximately 10.5%, which is slightly larger than our accuracy goal, but delivers much better results than the low-level SystemSim model. This suggests that more sophisticated modelling is required to accurately account for contention amongst the SPEs for the shared EIB and memory resources.

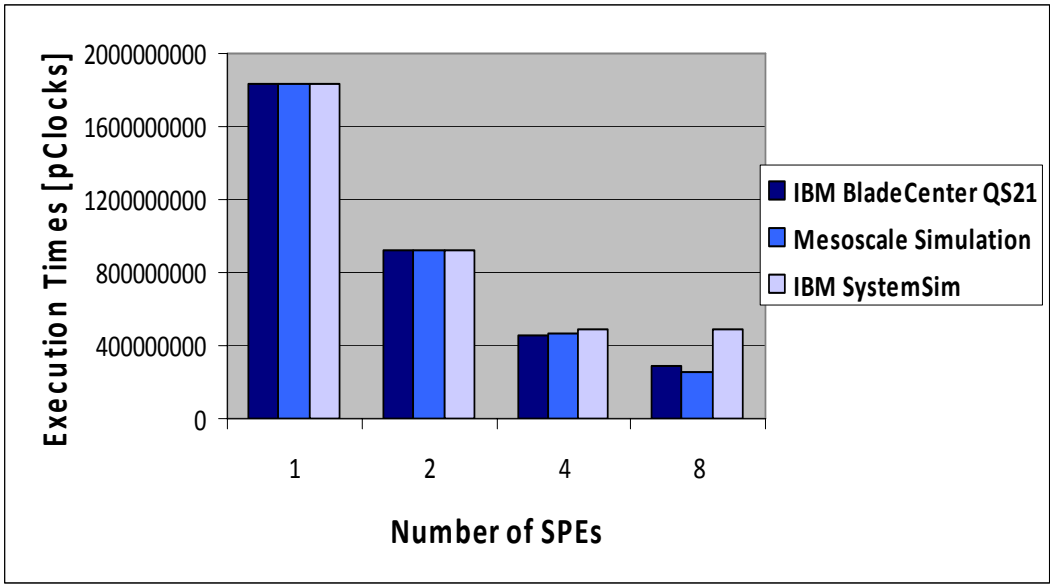

Figure 19: Simulation results and measurements of the FFT16M.

5.3 Execution Time

We measured the times for executing a FFT16M simulation using our mesoscale simulation infrastructure versus using the IBM SystemSim on the same server. As Figure

20 indicates, the mesoscale simulation is approximately 2 orders of magnitude faster. This is due to applying detailed simulations only to certain subsystems as described above.

On one hand this allows for a rapid design space evaluation pointing to design candidates that should be implemented in a full-system simulation environment like the IBM Full-System Simulator. On the other hand, the IBM Full-System Simulator provides a flexible and convenient environment to generate accurate workload characterizations for the mesoscale simulation. This demonstrates the synergies between these two modelling methodologies.

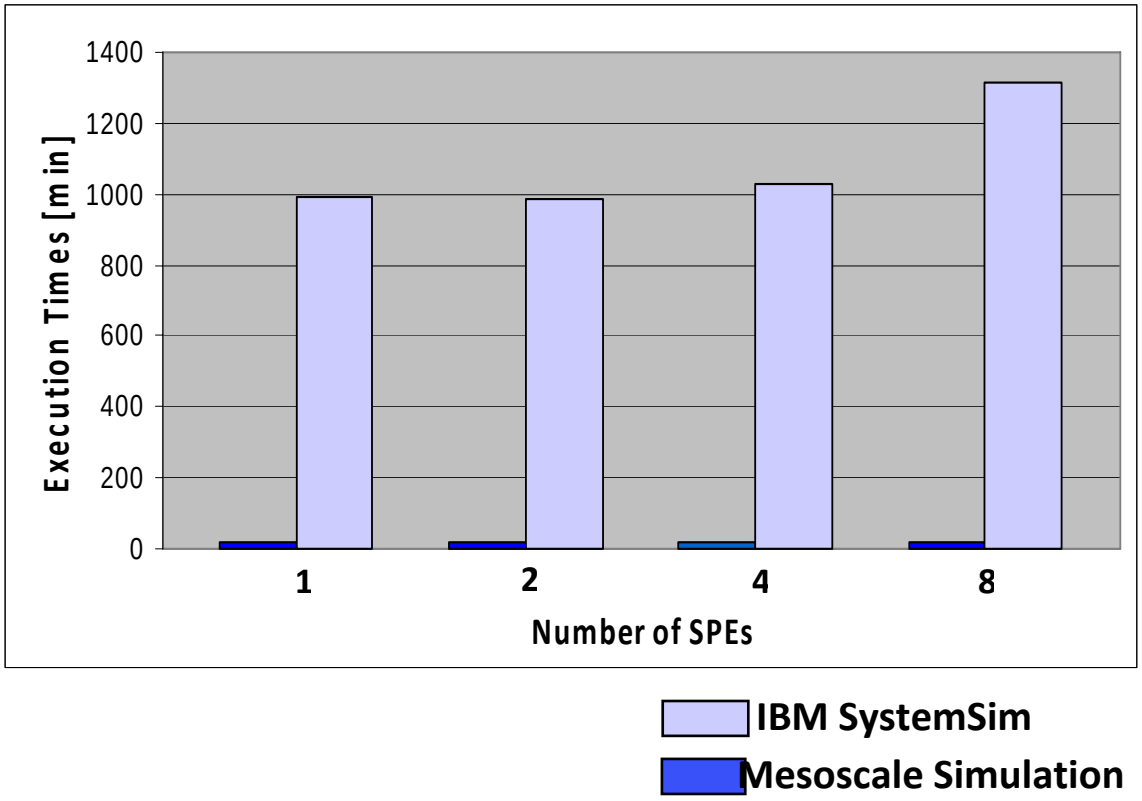

Figure 20: Execution times for simulating FFT16M for 1,2,4 and 8 SPEs mesoscale simulation infrastructure versus using the IBM SystemSim. The mesoscale simulations are approximately two orders of magnitude faster.

6 Conclusion

After an introduction to the method of mesoscale performance simulations, we have shown how this method can be applied to model the memory subsystem of the Cell/B.E. processor, a multicore processor with asynchronous memory access. We have demonstrated that the dual-approach strategy of mesoscsale simulations, i.e. modelling parts of the system and workload at different abstraction layers, can be highly efficient and accurate in these cases.

7 References

[1] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, D. I. August, Microarchitectural exploration with Liberty. Proceedings of the 35th International Symposium on Microarchitecture, November 2002.

[2] S. Önder, R. Gupta. Automating Generation of Microarchitecture Simulators. Proceedings of the International Conference on Computer Languages (ICCL) 1998.

[3] D. I. August, J. Chang, S. Girbal, D. G. Perez, G. Mouchard, D. A. Penry, O. Temam, and N. Vachharajani. UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development. IEEE Computer Architecture Letters (CAL), Aug 2007.

[4] AJ KleinOsowski and David J. Lilja, "MinneSPEC. A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. Computer Architecture Letters, Volume 1, June, 2002.

[5] Timothy Sherwood, Erez Perelman, Greg Hamerly and Brad Calder. Automatically Characterizing Large Scale Program Behavior. Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, October 2002.

[6] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. Proceedings of the 30th Annual International Symposium on Computer Architecture, June, 2003.

[7] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. TurboSMARTS: Accurate Microarchitecture Simulation Sampling in Minutes. Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, 2005.

[8] V. Pai, P. Ranganathan, and S. Adve. The Impact of Instruction-Level Parallelism on Multiprocessor Performance and Simulation Methodology. Proceedings of the International Symposium on High Performance Computer Architecture (HPCA), pp. 72-83, February, 1997.

[9] W. Lee, K. Patel. M. Pedram. B2Sim. A Fast Micro-Architecture Simulator Based on Basic Block Characterization. CODES+ISSS'06, Oct, 2006.

[10] E. S. Tam, J. A. Rivers, and E. S. Davidson. Flexible Timing Simulation of Multiple-Cache Configurations. Electrical Engineering and Computer Science Department Technical Report 348-97, University of Michigan, 1997.

[11] W. E. Denzel, J. Li, P. Walker, Y. Jin. A Framework for End-to-end Simulation of High-performance Computing Systems. Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops, March 2008.

[12] V. Pillet, J. Labarta, T. Cortes, S. Girona, PARAVER: A Tool to Visualize and Analyze Parallel Code, Technical report UPC-CEPBA 95-3. Available at: ftp://ftp.ac.upc.es/pub/reports/CEPBA/1995/. 1995.

[13] C. A. Prete, G. Prina, L. Ricciardi. A Trace-Driven Simulator for Performance Evaluation of Cache-Based Multiprocessor Systems. IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No. 9, Sep 1995.

[14] T. Wild, A. Herkersdorf, R. Ohlendorf. Performance Evaluation for System-on-Chip Architectures using Trace-based Transaction Level Simulation. Journal of Systems Architecture: the EUROMICRO Journal Volume 53, Issue 10 (October 2007).

[15] T. Isshiki, D. Li, H. Kunieda, T. Isomura, K. Satou. Trace-Driven Workload Simulation Method for Multiprocessor System-On-Chips. Design Automation Conference ’09, July 2009.

[16] E. Schnarr, J.R. Larus. Fast Out-Of-Order Processor Simulation Using Memoization. Proceedings of ASPLOS98 (San Jose CA, October 1998), 283-294.

[17] E. Ipek, S. A. McKee, B. R. de Supinski, M. Schulz, R. Caruana. Efficiently Exploring Architectural Design Spaces via Predictive Modeling. Proceedings of ASPLOS 2006.

[18] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, S. Huss-Lederman, M. D. Hill, J. R. Larus, D. A. Wood. Wisconsin Wind Tunnel II: A Fast and Portable Parallel Architecture Simulator. Workshop on Performance Analysis and Its Impact on Design (PAID), June 1997.

[19] G. Bloch, S. Greiner, H. de Meer, K.S. Trivedi. Queueing Networks and Markov Chains, Second Edition. Wiley-Interscience (2006).

[20] L. Kleinrock. Queueing Systems, Volume 1: Theory. John Wiley (1975).

[21] L. Kleinrock. Queueing Systems, Volume 2: Computer Applications. John Wiley (1976).

[22] A.M. Law, W.D. Kelton. Simulation Modeling and Analysis, Third Edition. McGraw-Hill (2000).

[23] J. Banks, J.S. Carson II, B.L. Nelson, D.M. Nicol. Discrete-Event System Simulation, Fourth Edition. Prentice Hall (2005).

[24] C.G. Cassandras, S.L. Lafortune. Introduction to Discrete Event Systems, Second Edition. Springer Science+Business Media (2008).

[25] J. A. Kahle , M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, D. Shippy. Introduction to the Cell Multiprocessor. IBM Journal of Research and Development, Volume 49, Issue 4/5 (July 2005).

[26] Cell Broadband Engine Architecture, Version 1.01, International Business Machines Corporation, Sony Computer Entertainment Incorporated,Toshiba Corporation (2005).

[27] Cell Broadband Engine Resource Center: Software Development Kit (SDK) 3.1. http://www.ibm.com/developerworks/power/cell/documents.html . Link last verified on 28.11.2009.

[28] IBM Full-System Simulator for the Cell Broadband Engine. http://www.alphaworks.ibm.com/tech/cellsystemsim. Link last verified on 28.11.2009.