COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF CRETE

Dynamic Dependence Analysis on Multi-core Processors

A.D.A.M.
(Accelerated Dependence Analysis for Multi-cores)

Master's Thesis

John Kesapides

(Ιωάννης Κεσαπίδης)

March, 2011

Heraklion, Greece

University of Crete

Computer Science Department

Dynamic Dependence Analysis on Multi-core Processors

A.D.A.M. (Accelerated Dependence Analysis for Multi-cores)

Thesis submitted by

John Kesapidis

In partial fulfillment of the requirements for the

Master of Science degree in Computer Science

THESIS APPROVAL

Author: _____

John Kesapidis

Committee approvals: _____

Dimitrios S. Nikolopoulos
Associate Professor, Thesis Supervisor

_____

Angelos Bilas
Associate Professor

_____

Manolis G.H. Katevenis
Professor

Departmental Approval: _____

Angelos Bilas
Associate Professor
Director of Graduate Studies

This work was performed at the Foundation for Research and Technology Hellas (FORTH), Institute of Computer Science (ICS), 100 N. Plastira Av., Vassilika Vouton, Heraklion, GR-70013, Greece.

The work is partially supported by the 7th European Commission Framework Programme through the ENCORE (FP7-STREP-248647), HiPEAC (FP7-NoE-004408), and HiPEAC2 (FP7-ICT-217068) projects.

Abstract

Recent trends in modern CPU architectures lead to multi-core designs with ever increasing numbers of cores. Furthermore architectures have emerged with heterogeneity and explicit memory hierarchies. The CELL [**1**] processor is a prime example of a powerful heterogeneous processor with explicitly managed local memories. A major challenge for such multi-core systems is the extraction of adequate parallelism and its exploitation with low runtime overhead and reasonable programming effort.

A suitable programming model for such architectures is the task-based model [**2**] [**3**] [**4**]. Task-based programming provides a high abstraction for the programmer while maintaining a significant amount of useful information. However the task-based model in far from panacea, since it requires explicit synchronization, which can be a limiting factor.

The dynamic data-flow execution model can overcome the bottleneck of explicit synchronization in task-based parallel programming, while at the same time simplifying the requirements from the programmer. Earlier research, such as the CELLSS [**2**] programming framework, proves that it is possible to implement the dynamic dataflow model over a task-based model via runtime dependence analysis. CELLSS [**2**] however performs expensive dependence analysis on memory objects by maintaining a task-graph at runtime. We propose an alternative method of implementing the data-flow model of execution over task-based models, where instead of a task graph we use a data dependency graph, and a novel mechanism for identifying dependencies with O(1) complexity. Furthermore our mechanism has the ability to track dependences due to partially overlapping data regions accessed by different tasks.

We design and implement ADAM, a runtime system that employees our proposals. ADAM stands for “Accelerated Dependence Analysis for Multi-cores”. We evaluate the scalability of ADAM on the CELL [**1**] processor and compare its performance with the (i)TPC [**3**] runtime, the (ii) CELLSS [**2**] runtime and (iii) a runtime with manual dependence analysis. ADAMs performance compared to the CELLSS [**2**] runtime is 2,77 better for the Cholesky benchmark and 2,27 times better for the Jacobi benchmark. In comparison with the TPC [**3**] runtime ADAM manages to efficiently parallelize applications that the TPC [**3**] fails to.

Acknowledgements

I would first of all to express my appreciation to my supervisor, professor Dimitris S. Nikolopoulos, and to professor Angelos Bilas, for their valuable guidance and patience. Their advise over the course of my Master's degree, were a major contribution to my evolvement as a researcher.

I would also like to thank the CARV laboratory of ICS-FORTH, for providing the necessary equipment, the research environment and stimuli. I am also grateful towards CARV laboratory, for the provided student scholarship, over the last two years. Also a special thanks to all the researchers, past and current, at CARV laboratory, especially to post-doc researcher Polyvios Pratikakis.

Finally a very special thanks to my family, my parents Panayiotis and Eleni, my brother Harry, and Fani for their support throughout my graduate and undergraduate studies.

Contents

List of Figures

List of Tables

1 Introduction

With the introduction of the power-wall, CPU architectures shifted towards the multi-core paradigm. In addition new heterogeneous architectures have emerged. Although the theoretical peak performance is constantly increasing, current programming models and tools are unable to utilize it, and as the number of cores increases they become more and more inadequate. Active research in the field of parallel computing has produced numerous proposals for programming models and new programming languages. One of the most promising, easy to use and efficient programming models is the task-based parallel programming model. From the programmer's perspective the task-based model abstracts the underline system thus allowing the programmer to express parallelism at the application level. From the systems perspective it provides the required information for the efficient parallel execution and communication of applications. Furthermore the constant increase of cores and the addition of small local-memories, reduce task-management cost and therefore favor the use of fine-grain tasks. Fine-grain tasks allow programmers to express parallelism easier and more efficiently.

An interesting architectural trend in modern multi-core processors is Heterogeneity. Heterogeneity can offer better performance and power efficiency but on the other hand it increases complexity, because it requires, tailoring program parts for each core type. Another architectural trend of particular interest is explicitly managed local memories. Explicitly managed local memories can improve performance significantly especially in the case of streaming applications, but delegating the responsibility of the transfers to the programmer makes them harder to program for. Task-based programming models help abstract the challenges in the programmability of such architectures.

The CELL [**1**] processor is a prime example of a powerful heterogeneous processor with explicitly managed local memories. The CELL [**1**] processor consists of nine cores, one dual-threaded 64-bit PowerPC called PPE(Power Processing Element) and eight 128-bit SIMD processors called SPEs(Synergistic Processor Element). Each SPE has a 256KB local storage for data and code that is not implicitly coherent with the main memory. The programmer is responsible for fetching and writing-back the appropriate data and code via DMA commands. Although the peak performance of this processor is highly promising, the

heterogeneity and the explicit coherency among cores make it extremely hard to program for.

The task programming model proves to be a good abstraction for the CELL [**1**] processor and can greatly simplify the development process, because it can abstract heterogeneity to a certain degree and it can implicitly express data transfers. The PPE is defined as the task arbitrator and the SPEs as the task executors. The CELL [**1**] processor's architecture favors a centralized model with one master and multiple workers and over the years several runtime systems and languages have been proposed that employ this model or some variation of it [**2**] [**3**] [**4**]. Tasks implemented over the CELL [**1**] architecture are defined as self contained remote and asynchronous functions consisting of both code and data.

Although the task programming looks sufficient it does present certain issues. The programmer is responsible for the synchronization among tasks, which in certain situations is not trivial, unless the programmer is willing to sacrifice performance. Furthermore synchronization often impedes scalability, and in certain cases the task-model, as is, cannot properly express parallelism in applications. Let us consider the following case. An iterative application where each iteration consists of three phases, init, calculate and complete. Each phase is sufficiently parallel but every task in the calculate phase depends at least on one task of the init phase and every task in the complete phase depends at least on one task of the calculate phase. The dependencies among tasks are a result of the current state of the application and so there are no guaranties that two iterations will exhibit the same dependencies. In order to use the task programming model we would introduce synchronization after each phase of each iteration. There is however additional parallelism we do not exploit. Task executions could be pipelined among the three stages of each iteration, as well as across iterations.

These issues grow in importance as the number of cores increases. CPU designs have already emerged with as many as 48 cores [**5**], and synchronizing among 48-cores is bound to impact performance. Furthermore multi-core designs, are now becoming the mainstream, which means that the target group for these CPUs in longer just the experts of the high-performance community but rather the average programmer. In addition it is not uncommon for software intended for the average user, to be dynamically parallel, like video-games for instance. An application is considered dynamically parallel when the amount of parallelism cannot be statically identified a priori.

A solution to the issues above would be a dynamic dataflow-model of execution, because this model does not require any explicit synchronization from the programmer and it inherently expresses parallelism. The data-flow model [**6**] dictates that an operation should proceed only when, all the values it depends on are updated. Dependencies among operations in the data-flow model are expressed by a directed acyclic graph [**6**]. **Figure 1—1** shows an example derived from [**7**] of a dataflow graph that corresponds to a small program.

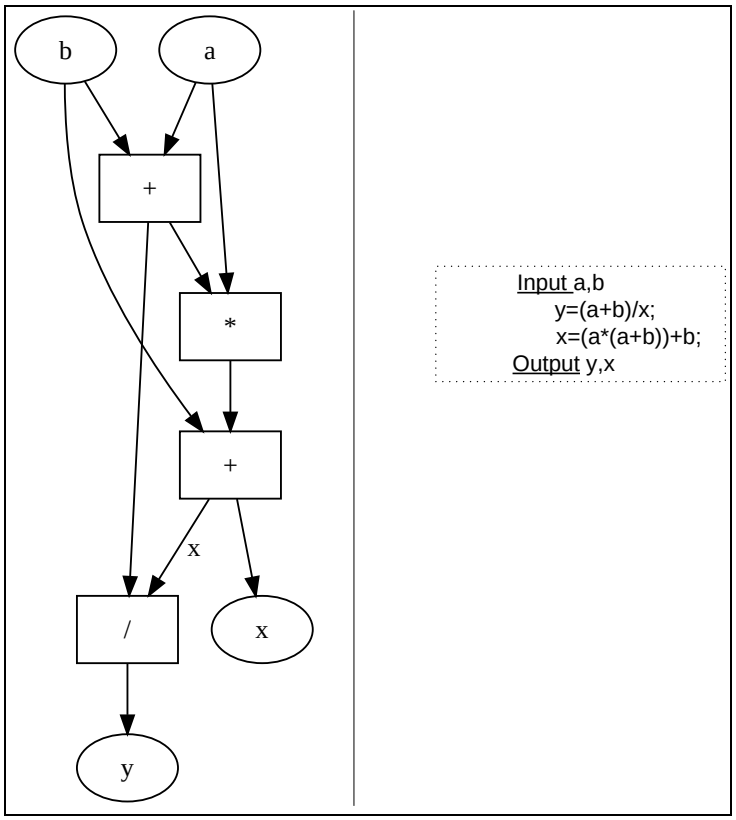

Figure 1—1 Dataflow example from [7].

The CELLSS [**2**] [**8**]runtime proved that it is possible to use the data-flow execution model over a task-based model, using tasks as the operations and the data-transfer directions as access descriptors (Read, Write or both). Because CELLSS [**2**] is a runtime system, the directed acyclic graph of tasks, representing dependencies, is created and respected dynamically, at runtime. Although this promotes parallelism, in order to create the task graph the runtime for every newly created task has to search within the set of resources used by other tasks in order to find dependencies. Therefore the management of the task graph introduces overhead typically in the order of thousands of core cycles [**8**].

We propose an alternative method of implementing the dynamic data-flow model of execution over task-based models. Our method presents a significantly lower overhead compared to task-graph solutions. Since dependencies concern data and operations on data, instead of a task graph we use a data dependency graph. In this graph the edges are operations on data and the arcs are tasks (execution paths). Because the dependencies that emerge among data operations are not all true dependencies, we can rename the data thus resolving them. Data renaming is a technique used in resolving register dependencies within CPUS, and is also used by the CELLSS [**2**] runtime. In our solution we consider that the runtime always resolves all but the true dependencies. This allows us to simplify our data-dependency graph since we only need to express Read-after-Write dependencies. It is sufficient in order to represent Read-after-Write dependencies to have only write operations represented as edges in our graph. So our graph is a data-dependency graph with write operations on data as edges and tasks as the arcs connecting the edges. This graph is usually less complex than the equivalent task-graph and easier to manage. Furthermore because our dependency graph is a data-dependency graph and not a task graph, the renaming process does not require additional address translations. Two write operations on the same data will be represented as two distinct nodes in the graph, therefore once dependence analysis takes place, we do not have to perform additional look-ups to accommodate renaming with the correct versions of the data.

We present a new algorithm for detecting dependencies with O(1) cost and the ability to track dependences due to partially overlapping data regions accessed by different tasks. We view the entire memory as a matrix of aligned blocks, the size of which is a configurable parameter. From the perspective of dependencies we consider that all the operations occur on units of memory blocks. This induction allows us to detect dependencies among overlapping data regions. Whenever a new write operation is issued we translated it into memory blocks and use the first few bytes of each block to store a unique-id that uniquely identifies a corresponding node in the graph. In order to detect dependencies, for a specific block we just look at the beginning of the block for a valid id.

We implement and present the design of a runtime system called ADAM. ADAM stands for Accelerated Dependence Analysis for Multi-cores. ADAM was primarily intended for the CELL [**1**] processor, therefore the design of the runtime is driven from the architecture of the CELL [**1**] processor. The starting point for the development of ADAM was the TPC [**3**] runtime system and because of this, ADAM is compatible with the TPC [**3**]. TPC

[**3**] is a task-based low-overhead runtime system for the CELL [**1**] processor. Although the target architecture for ADAM is the CELL [**1**] processor, we have also implemented an SMP version for x86 multi-core processors, we describe the design differences of the SMP version in a separate sub-section within the design section.

We evaluate the performance of ADAM in a series of application from the SPLASH-2 benchmark suite, the CELLSS [**2**] runtime and the sequoia runtime. We also compare the performance of ADAM with the (i)TPC [**3**] runtime, the (ii) CELLSS [**2**] runtime and (iii) a runtime with manual dependence analysis. ADAM exhibits good scalability in cases where the tasks are large enough to outweigh the introduced overhead. In the cases where ADAM exhibits good scaling, ADAM's performance is always better than that of the TPC [**3**], and in all of our evaluation cases ADAM outperforms the CELLSS [**2**] runtime. ADAM is 2,77 times faster for the Cholesky benchmark with 13x13 dataset and 2,27 times faster for the Jacobi benchmark, from CELLSS [**2**] in runs with 6 SPEs. Compared to the TPC [**3**] runtime ADAM is 2,43 times faster for the Jacobi benchmark and 5,99 for the Matmul benchmark.

2 Design

ADAM uses the model of a single master and multiple workers, driven from the architecture of the CELL [**1**] processor, with the master and the workers each with its own address space. The master issues tasks to the workers and performs dependence analysis, while the workers only execute tasks. Each worker has its own queue visible only to him and the master. The master pushes tasks to the worker's queue and workers execute them. The worker is responsible for fetching and writing-back the data necessary for each task. TPC, which ADAM uses as a back-end allows for the data transfers of one task to overlap with the execution of another.

We consider tasks a set of self-contained remote asynchronous functions. Tasks may call other functions and allocate/de-allocate data but all interaction with shared data among other tasks and the main thread must be stated upon task creation. The programmer must pass the shared data as arguments to that task and denote whether this task intends to read, write or both read and write on each argument separately. We will refer to the intent of this denotation as an "operation" from now on. Tasks are also not allowed to create new tasks. Tasks are asynchronous since their execution can commence at any point in time after their dependencies are satisfied.

2.1 Detection/Registration

For every issued task, ADAM needs to perform dependence analysis for it. ADAM handles each argument separately, translates it into its corresponding memory blocks and performs dependence analysis for each block. The size of the blocks is a parameter defined by the programmer at the initialization of the runtime and represents the granularity of the dependence analysis. The programmer can also flag a specific argument as "safe", in which case no dependence analysis is performed for this argument nor is it divided into blocks. For stride-arguments each stride element is treated as a separate argument. The translation of arguments to blocks allows us, to detect dependencies among partially overlapping arguments since the overlapping ranges will translate to the same blocks.

Detection of dependencies consists of keeping track of addresses with incomplete operations and searching whether each newly issued address belongs to these addresses. Therefore we need low-cost search within our data-structures as well as low cost insert. A common alternative is the use of hash tables, but address-based hashing will present

collisions for every dependence. In ADAM we alleviate the search operation using the following method.

To detect dependencies we use the first two bytes (2byte-ID) of the block and the address of the block and create the tuple {2byte-ID,address}. The runtime preserves metadata per operation in an array. Each metadata element in this array has a field that contains the address it is associated to. We use the first element (2byte-ID) of the tuple as an index for this array and compare the second element of this tuple (address) with the address this metadata element is associated to. If the two addresses match then we have found a dependence. The reason that we also need to compare the addresses is that the first two bytes of the block may be user data and not a 2byte-ID placed by ADAM and we need a way to exclude these cases as false detections.

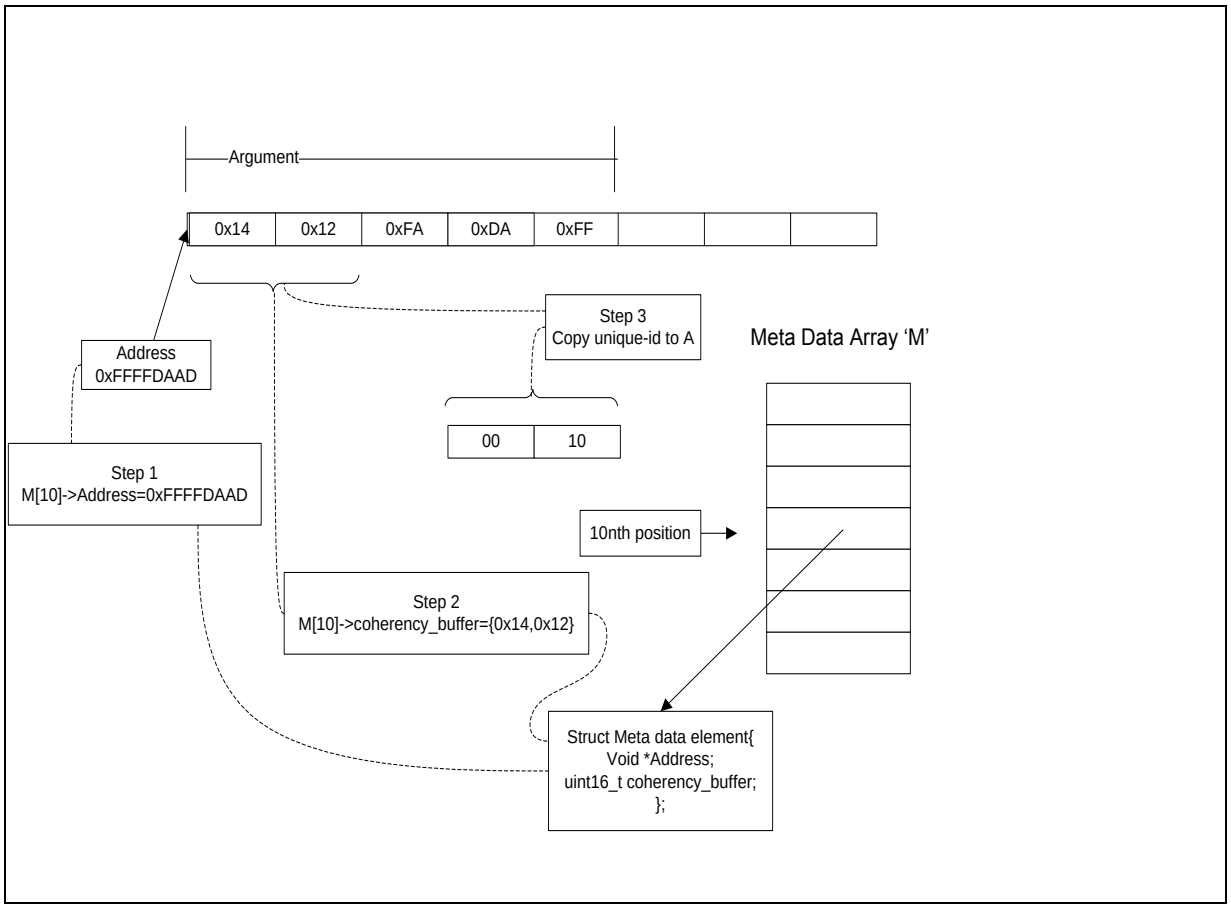

Figure 2.1—1 Dependence Analysis Register. Example of the Register operation for a specific block.

In order to register(**Figure 2.1—1**) an operation to a block by a task we do the following: First we reserve a new slot in the metadata array and a new meta-data element. We then set the field of the metadata element's associated address to the address of the block we are registering. Finally, we buffer the first two bytes into a field in the meta

element called coherency_buffer and overwrite them with our index (2byte-ID). For each block we perform dependence analysis, we always first detect and then register. If a block is already registered then instead of placing the first two bytes of the block in the coherency buffer we simply copy the coherency buffer of the already registered meta-data element. The rest of the process remains the same. Multiple elements in the metadata array are allowed to be associated with the same address but detection guarantees the return of the one that was issued last because the first two bytes of the block will always be the index of the last one. It suffices to only register write operations on blocks because only Read-After-Write dependencies are true dependencies and have to be respected by the runtime, the rest can be resolved using renaming. Therefore for read operations on blocks we only detect not register.

2.2 Dependency Model

The runtime handles Data-dependencies among tasks. These dependencies can expressed by a slightly modified Bernstein Condition:

$$[I(T_i) \cap O(T_j)] \cup [O(T_i) \cap I(T_j)] \cup [O(T_i) \cap O(T_j)] \neq \emptyset$$

Where:

- i and j express the issue order of the tasks and i<j
- $I(T_x)$ is the set of blocks read by task T_x
- $O(T_y)$ is set of blocks written by task T_y.
- There is a valid run-time execution path from T_i to T_j
- No explicit synchronization exists among the two.
- j-i<Task_Window. Task window is the maximum number of allowed issued tasks in the runtime at any single point in time.

These dependencies are divided into three categories:

a. True Dependencies formally expressed $O(T_i) \cap I(T_j)$ with the above preconditions
b. Anti-Dependencies formally expressed $I(T_i) \cap O(T_j)$ with the above preconditions
c. Output Dependencies formally expressed as $O(T_i) \cap O(T_j)$ with the above preconditions.

In order to keep track of dependencies ADAM does not maintain a task graph but rather a data-dependency graph. In this graph data operations on addresses are the nodes of the graph while tasks are treated as arcs. As mentioned above though the runtime system respects only true dependencies and therefore only registers write operations. The data-

dependency graph allows us to handle only true dependencies as opposed to a task graph. Furthermore, because, dependence detection takes place over data we reduce the control path required to associate detected dependencies with the corresponding tasks. A data dependency graph can also be used for the purposes of renaming thus reducing the control path even more. In addition task management is greatly simplified due to the fact that a dependency graph disassociates the dependence management from tasks.

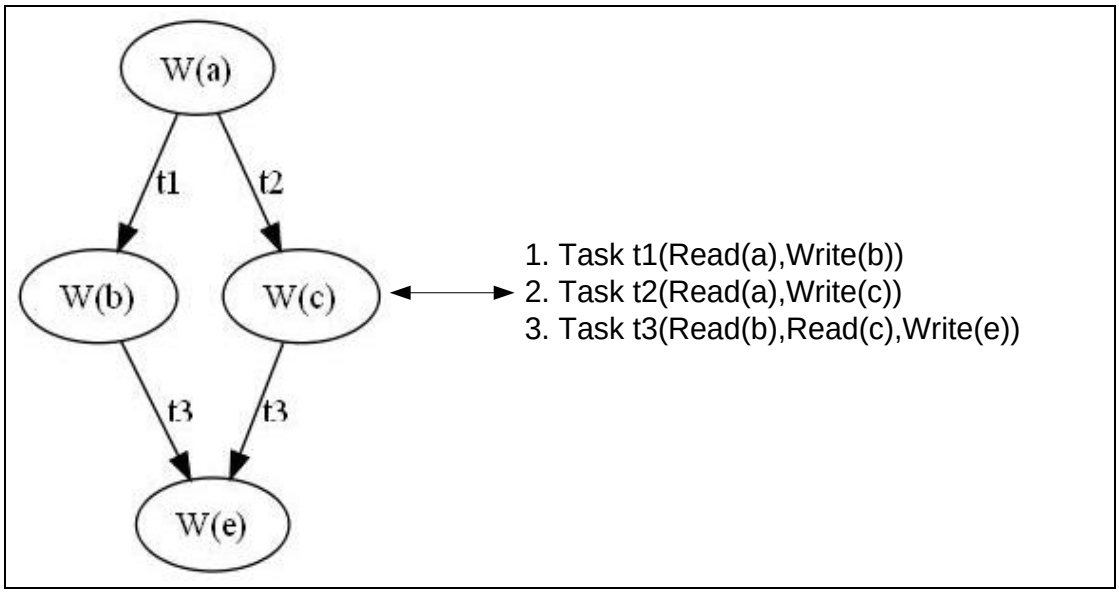

Figure 2.2—1 Data dependency Graph. Example of a data dependency graph along with the task issues that correspond to this example.

Writes are the nodes of the graph and tasks are directed arcs connecting them. A task (arc) connects two writes if and only if the task reads the value of the source write (edge at the beginning of the arc) and performs the destination write (edge at the end of the arc). Since the same task may present itself as more than one arcs, we consider each task an arc-class and the arcs for that task as instances of that class. The default behavior of the runtime system is to treat class-instances as a unity (all these paths will be created together and progressed together), the dependency model however allows you to treat them separately (progress a path earlier/later) and to add/drop class-instances dynamically as long as the arc-class is alive. An arc-class is considered alive for as long as it has one or more instances.(Current implementations of the runtime impose the default behavior though).

2.3 Task Management

Tasks that have dependencies need to be blocked from executing until their dependencies are resolved. Because a task is an arc in our dependency model we need only

to keep track the number of pending dependencies. The task has no information about which dependencies are pending but only how many dependencies are pending. When the number of pending dependencies reaches zero then the task is eligible for execution. For each Write operation on a block we preserve a meta-data element. Each element has its own stack of references to tasks that depend on it. When we create a task if the analysis detects a block registered then this means that the task must block until the registered operation completes. Therefore we push a reference to the task we are currently issuing to the stack of the meta-data element of this operation and increment the pending dependencies for the task by one. The meta-data for the tasks also include a full description of the task. These include a function-id, which identifies the function to be called, and for each block an input address, an output address, the size of the block, an offset within the block, and a flag denoting whether we read, write of both to the block, and whether this is the first block of the original argument.

For every task that completes execution we need to update the data-dependency graph. For each meta-data element, that the completed task registered, we pop empty the stack of dependent tasks. For every task that we pop, we decrement its pending dependencies counter by one. If for any of the popped tasks this counter reaches zero then this task is considered eligible for execution. We will describe later how we link meta-data elements with the tasks that created them(**2.4 Timing Model**), as well as what happens to a task when it is eligible for execution(**2.7 Scheduling**).

2.4 Timing Model

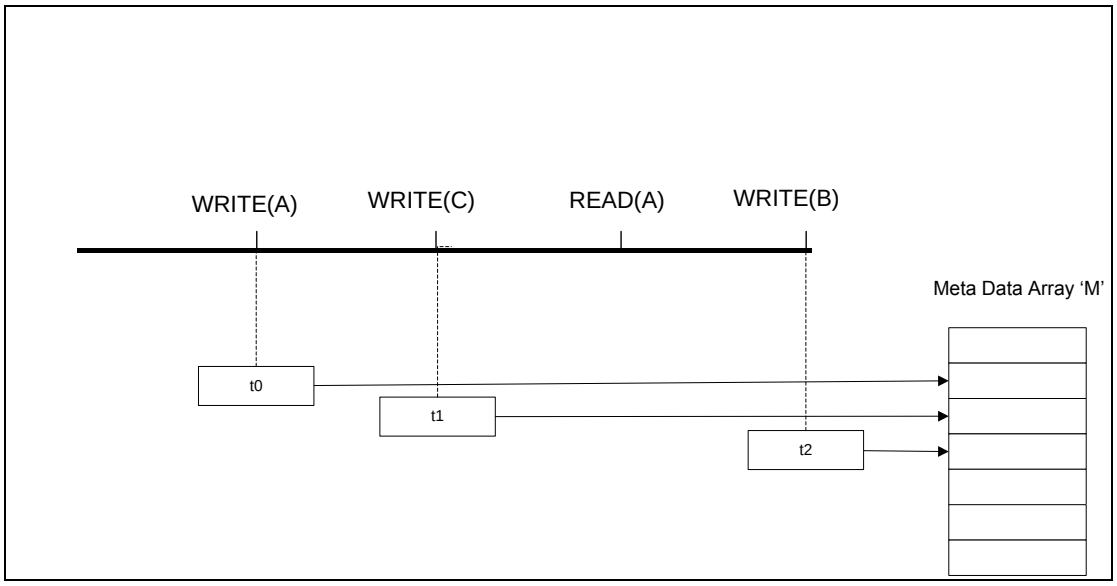

Figure 2.4—1 Timing Model

The timing model is used to imposed relative ordering in the system. In our design we are required to have knowledge of the oldest issued write (**2.6 Meta-data Management**), and precedence among writes in various implementation corner cases. The system maintains a single centralized logical clock [**9**] called "Epoch". Events that signify time progression in our system are considered write operations. Tasks and meta-data elements get time stamped with the current clock value when they are issued. Since "Epoch" is a monotonically increasing value and registered meta-data elements represent writes each registered element has a unique clock value. We use this clock value as the index to the meta-data array and subsequently as the 2byte-ID used for detection/registration. The meta-data array can now be viewed as the applications timeline. The value of the clock may exceed the size of the array so it is normalized to fit in it, which means that, as logical time progresses we cycle through the array. Tasks get time-stamped prior to the analysis of their arguments. The logical clock value for the task is the time base for its registered elements, so if a task has logical clock value T then its registered elements will have logical clock values T+1, T+2, …,T+N for N registered elements by this task. This means that the registered elements of this task will reside on the normalized range [T+1, T+N] of the array. For each task we issue, we have a dedicated field on its meta-data that counts the number of issued writes (the number of registered elements) and a dedicated field for the task's time-stamp. When a task is completed we use these two fields to deduce which meta-data elements

were issued by the task and update them. Time relations among tasks are not respected by the runtime, because the dependency model ensures correctness in execution. Enforcing happened before in the execution order of tasks would restrict severely the parallelism of the application.

2.5 Renaming

ADAM performs renaming to resolve Write-After-Write and Write-After-Read dependencies. ADAM has a policy of always renaming and uses a set of pre-allocated buffers of block-size for renaming. This policy simplifies the dependency graph since only true dependencies are respected. Renaming consists of simply changing the target of a write operation for a task. Write operations are represented by meta-data elements. These meta-data elements contain a field that points to the renamed buffer for this write. When a task is issued and dependencies are detected we change the task's reads and writes to the renamed buffers if necessary. If we are issuing a read to a block then if it is registered we change the read address to the renamed buffer associated with the registered meta-data element, otherwise it remains unchanged. If we are issuing a write then we change the write address to the renamed buffer associated to the meta-data element we are registering. If we issue a read/write to a block then we treat it first as a read and then as a separate write. Renamed buffers are written back lazily. When multiple writes are issued to a block only the last one is eventually written back. Whenever a write is issued to a block then if this block is already registered then we add a link to from the new meta-data element to the previous meta-data element. We keep linked lists between meta-data elements associated to the same block. Only the renamed buffer of the head of this list is eventually written back the rest are considered intermediate values. To avoid violating resolved Write-After-Read dependencies writing back is restricted to the oldest issued write in the system at a time, because the oldest issued operation by definition cannot be preceded by an unfinished read operation.

Because the meta-data elements are used for the purposes of renaming they have to follow the lifetime of their associated renamed buffers. This means that meta-data elements may stay registered after a task has completed. We define two distinct phases for meta-data elements called "Active" and "Inactive" accordingly. A meta-data element starts as "Inactive" and when a task is completed and the stack of the meta-data elements dependent tasks is popped empty the meta-data element is set to its "Active" state. For

each block that we detect dependencies for, we also check the state, of the registered element if any. If a registered element is Active then we do not push the task into the element's stack and we do not change the tasks pending dependencies counter, we simply redirect our issued task's block read or write to the renamed buffer. If it is in its "Inactive" state we proceed as normal (as described earlier).

2.6 Meta-data management

The meta-data elements, the rename-buffers and the size of the meta-data array are pre-allocated and finite in count, therefore throughout the execution of an application we collect and reuse them. In order to collect a meta-data element and the rename buffer associated with it, we first need to make sure that all readers to this write operation are completed. For this purpose each meta-data element has a counter called data_beggers_counter. While the meta-data element is in "inactive" state this counter is not used. During the meta-data elements "Active" state each pending reader increments this counter and each completed reader decrements it. As previously described when a task completes then for each of its meta-data elements, it pops their stacks empty. Each of these meta-data elements is then set to its "Active" state. At this point it also initializes the data_beggers_counter to the number of tasks popped from the stack, also for every read, issued to a registered element, that is in its "Active" state we increment that element's data_beggers_counter by one. Each task maintains a small vector with the indexes of the registered elements that it issued a read operation to. When a task completes it uses the indexes from this vector to revisit these meta-data elements and decrements their data_beggers_counter by one. If a registered element is in "Active" state and its data_begger_counter equals zero then it is eligible for collection.

There are two distinct methods of collection within the runtime denoted as soft collection and hard collection. Soft collection takes place every time the dependence detection finds a registered element. Registered elements that are associated with the same block are linked via a linked-list, with the head of the list being the last issued element and the rest being intermediate values. Intermediate values do not require write-back to the original address, so soft collection iterates through the elements of this list. Starting from the second element, it checks each element if it is eligible for collection and acts accordingly.

Registered elements are placed on an array according to a logical clock(epoch). The array is bound in size whereas the clock not, and hence at some point we will collide with a

reserved array slot. Because the clock increases monotonically the element we will collide will be the oldest issued write at that time. Hard collection ensures forced collection of this element in order to place the new element at its place. If this element is no yet “active” then hard collection blocks and calls the scheduler, until it becomes “active”. If it was “active” or after it becomes “active”, it checks whether this element is an intermediate value. if it is an intermediate value it simply collects it otherwise it writes the renamed buffer back to the original and then collects it.

2.7 Scheduling

The scheduling policy is configurable depending on the specific needs of the running application. The first configurable parameter is the Task Window. The Task Window is the maximum number of issued and uncompleted tasks at a time. A larger window allows the runtime to potentially explore more parallelism by increasing the distance of tasks it can reorder. The Task Window Threshold is the number of tasks the runtime has to execute and complete in the event that the Task Window becomes full. Another parameter is the Multiplier. The Task Window times the Multiplier defines the number of meta-data elements that are allowed in the runtime as well as the size of the meta-data array. The Multiplier represents the lifetime of the renaming in units of Task windows. The default value for this parameter is one but a larger value can potentially improve performance by avoiding writing-backs.

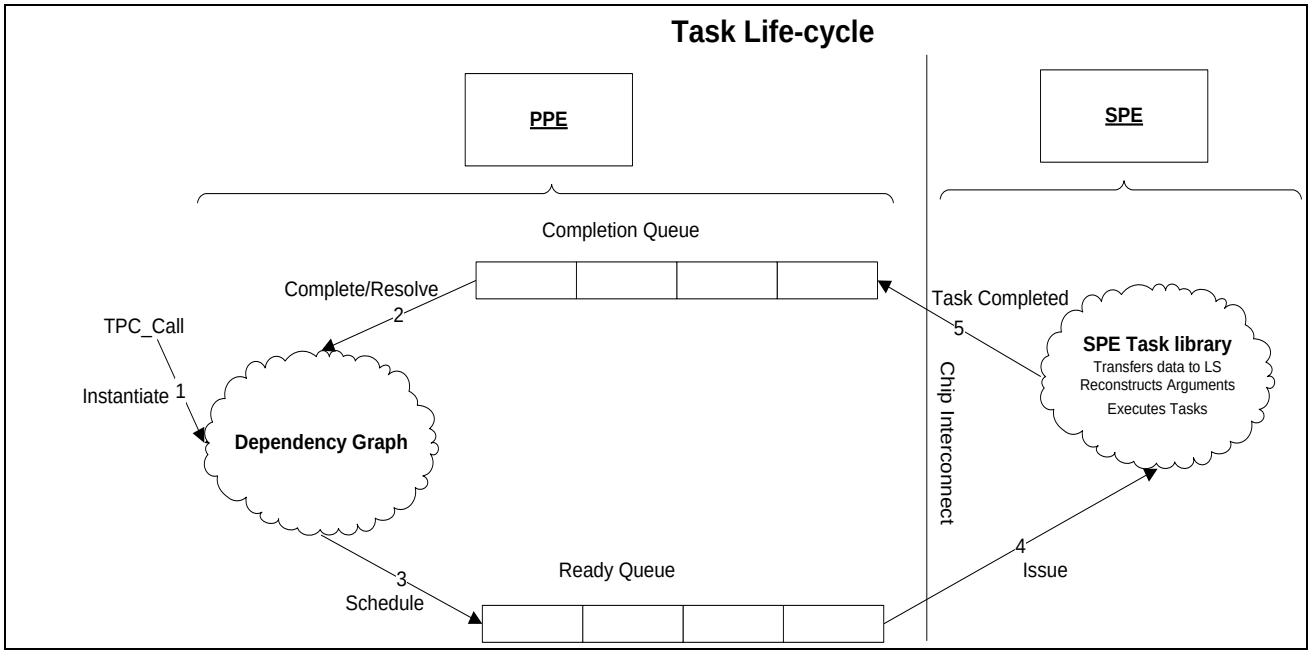

Figure 2.7—1 Task life cycle. The path of a task along the execution of an application

Tasks that are eligible for execution are pushed into a ready queue. The ready queue can be configured as an actual queue or a stack, which corresponds to breadth-first execution and depth-first execution accordingly. The scheduler component of the runtime de-queues tasks from the ready queue and assigns them to the workers. The master contains a completion queue for each worker, where the worker signifies task completions. The scheduler polls the worker's completion queues for an available worker in a round robin fashion. The worker that is available probably completed one or more tasks, if at this point the ready queue is empty we release any completed tasks of this worker. This lazy release policy reduces the scheduler lags thus yielding better scheduling efficiency. If the ready queue has at least one task we send a task to this worker, and release the task that previously resided on the now completed slot unless this task has already been released. Because the Master and the workers have different address spaces and the meta-data reside on the Master, the master keeps a two dimensional array called spe_map that maps task meta-data to tasks residing on the workers queue. This way when a task at specific queue slot of a worker finishes the master can look up the spe_map for the completed task's meta-data.

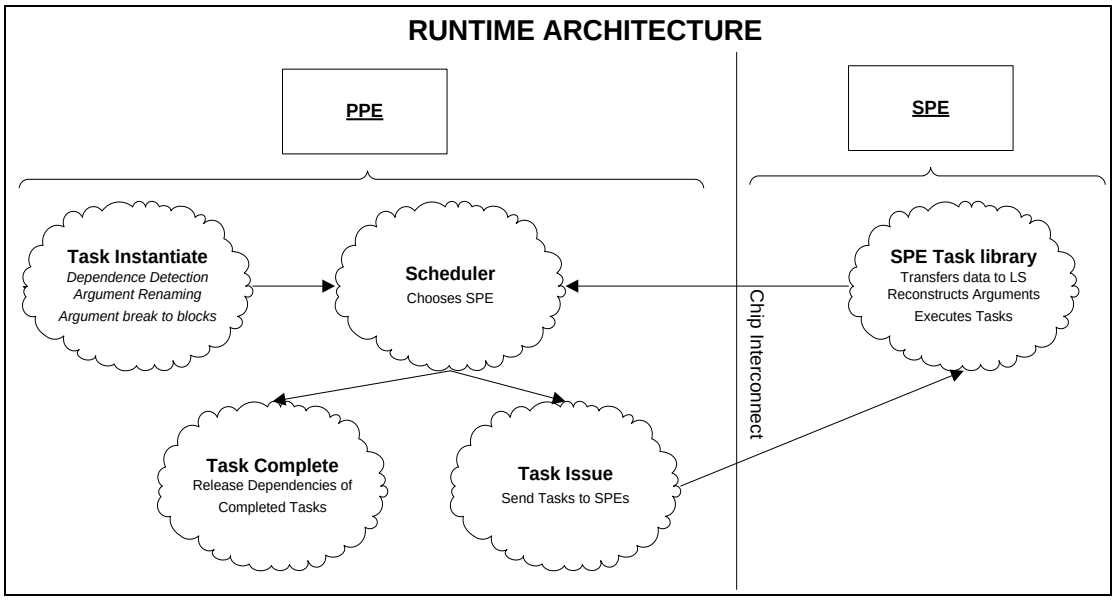

Figure 2.7—2 Runtime component Architecture. This figure depicts the main components of the runtime, and the interactions among them.

The scheduler is executed by the master synchronously at two points, before issuing a task and at barriers. At a barrier the scheduler executes until every issued task is executed and released. Before issuing a task the scheduler checks whether the Task window is full, if it is then it executes and releases tasks until it reaches the Task Window Threshold. The

default value for this threshold is the number of workers multiplied by the number of their queue size. The queue size of the workers is also a user defined parameter. If the task window is not full then scheduler executes tasks for as long as there are tasks in the ready queue and no stall is detected. While the scheduler polls the worker queues in a round robin fashion, if after one round no worker is free we consider that a stall is detected. At the scheduler invocations that stall detection is not a blocking precondition, the scheduler looks if the next task in the ready queue has an equivalent task, defined for the master. If there is an equivalent the master executes the task, releases it and re-polls the worker queues.

2.8 Consistency

ADAM breaks arguments into blocks and overwrites the first two bytes of each block. This operation however is abstracted from the programmer. The tasks need not take into account the blocking or the id-placing. The worker is responsible for re-constructing the arguments, before the execution of the tasks. When an argument is broken into blocks by the master, the master flags the first block as the start-block. The worker places the blocks in his local memory in the sequence they were issued and considers the local addresses of the start blocks as the local task arguments.

Prior to sending a task to a worker for execution, for every read block-address associated with the task we detect whether a meta-data element is registered to this block. If a meta-data element is registered then we get the two overwritten bytes from the meta-data element and pass them to the worker, who in turn after fetching the data for a task and prior to executing it places these two bytes at the beginning of the local copy of the block.

In certain cases the Master needs to use data that are written or read by tasks. This introduces two issues: (i) Synchronization in order to avoid races with tasks and (ii) cohesion in case the data have been renamed. For this purpose these accesses are performed through the runtime with the use of a special macro that given a specific address ensures that tasks associated with the block containing this address have completed. The macro also ensures that if this block is renamed the correct buffer will be used. The latter is particularly useful because the default behavior of the buffer ensures only the execution of all issued tasks and not the coherence of the memory. Coherency is guaranteed from the use of the macro. There is a special barrier that ensures memory coherency called Mem_sync that writes back all the renamed buffers and alleviates the need for the use of the macro after the barrier but it is less efficient than the default barrier.

2.9 ADAM-SMP

The dependency model of ADAM is architecture independent, the entire runtime design however is tailored for the CELL [**1**] processor. The greatest challenge in SMPs architectures regarding ADAMs design is coherency. Because the workers and the master share a single address space we cannot register and then reconstruct the data, without damaging either the consistency of the dependency graph either the consistency of the user data. Therefore we detach the dependency detection and registration form the user data while still maintain the good property of O(1) dependence analysis.

The SMP version of ADAM divides the memory into segments called grids. Grids are of the same size and alignment (like pages) and each of them is further divided into blocks. A grid can be thought of as an one-dimension array of blocks. The size of grids and blocks is an application-specific parameter, which can be tuned to trade dependence analysis overhead for additional false positive dependencies. All users allocations are encapsulated through a custom allocator (**Figure 2.9—1)**. The allocator internally has at least one Grid allocated. Starting from the end of the Grid the allocator translates the user request into units of ADAM blocks, and serves the user's request with memory in a stack like manner. The allocator also considers that starting at the beginning of the Grid there is an implicitly defined integer matrix with as many elements as the number of allocated blocks within the Grid. This integer matrix and the user's allocations compete for the same space(like heap and stack), therefore whenever the allocator detects and imminent collision, it allocates a new Grid and serves the subsequent requests in the same manner from the new Grid.

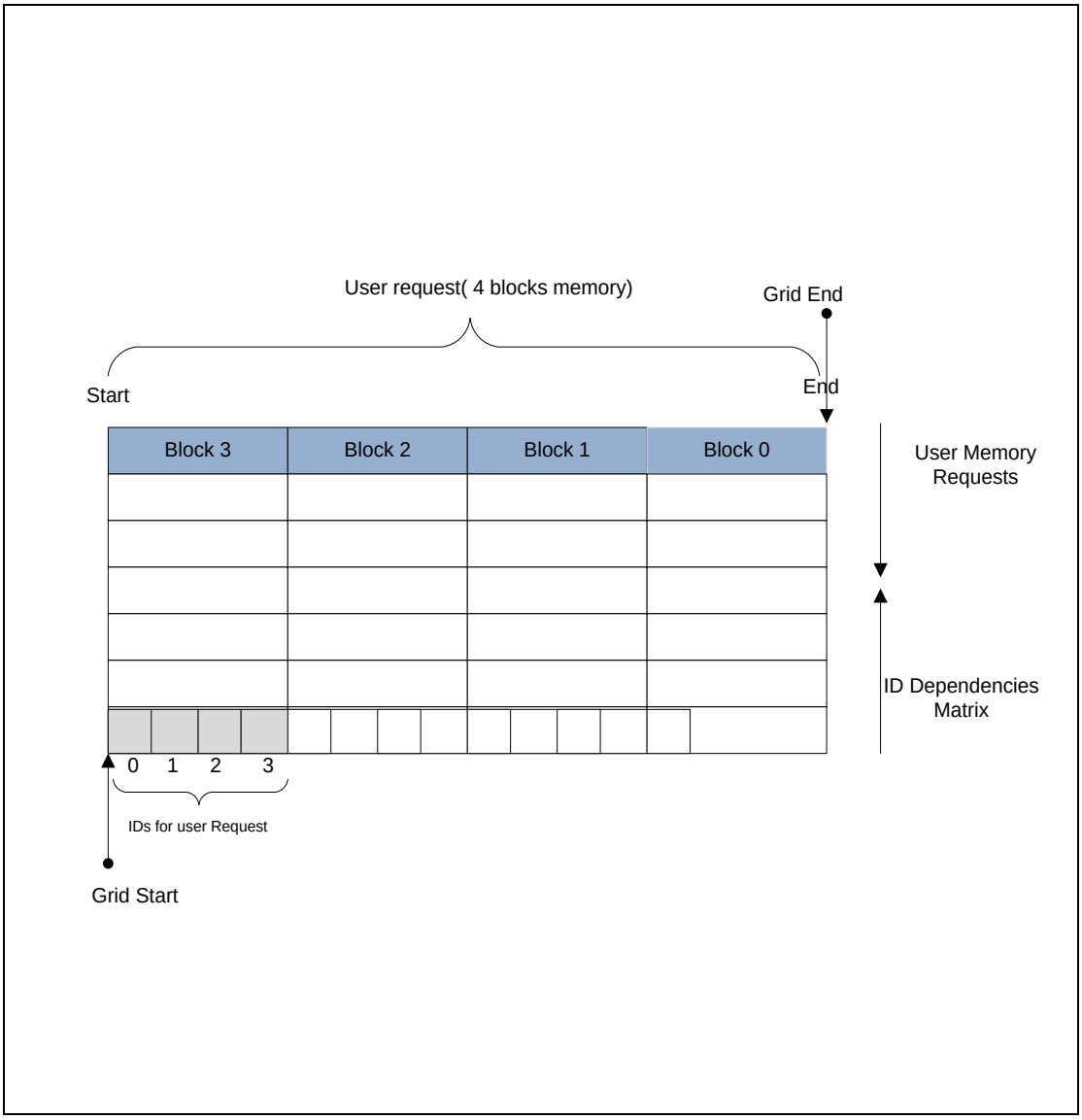

Figure 2.9—1 ADAM Grids. Allocation Example with the usage of Grids

Grids are aligned, therefore, for any given address in a grid we can identify the beginning and the end of the grid using fast bit masking operations. A grid is viewed as a matrix of user data blocks from end to start or as a matrix of unique task IDs from start to end. We use a custom memory allocator that allocates memory for user data in units of blocks from the end to the start of a grid. The relation between a block and a unique ID is the following: For a given block, we start from the end of the grid and measure the distance in blocks (e.g. the 3^{th} block from the end). Starting from the beginning of the grid we index the grid as an integer matrix to the aforementioned distance for the desired task-ID**(Figure 2.9—2)** (e.g. ((int *)G)[2]).

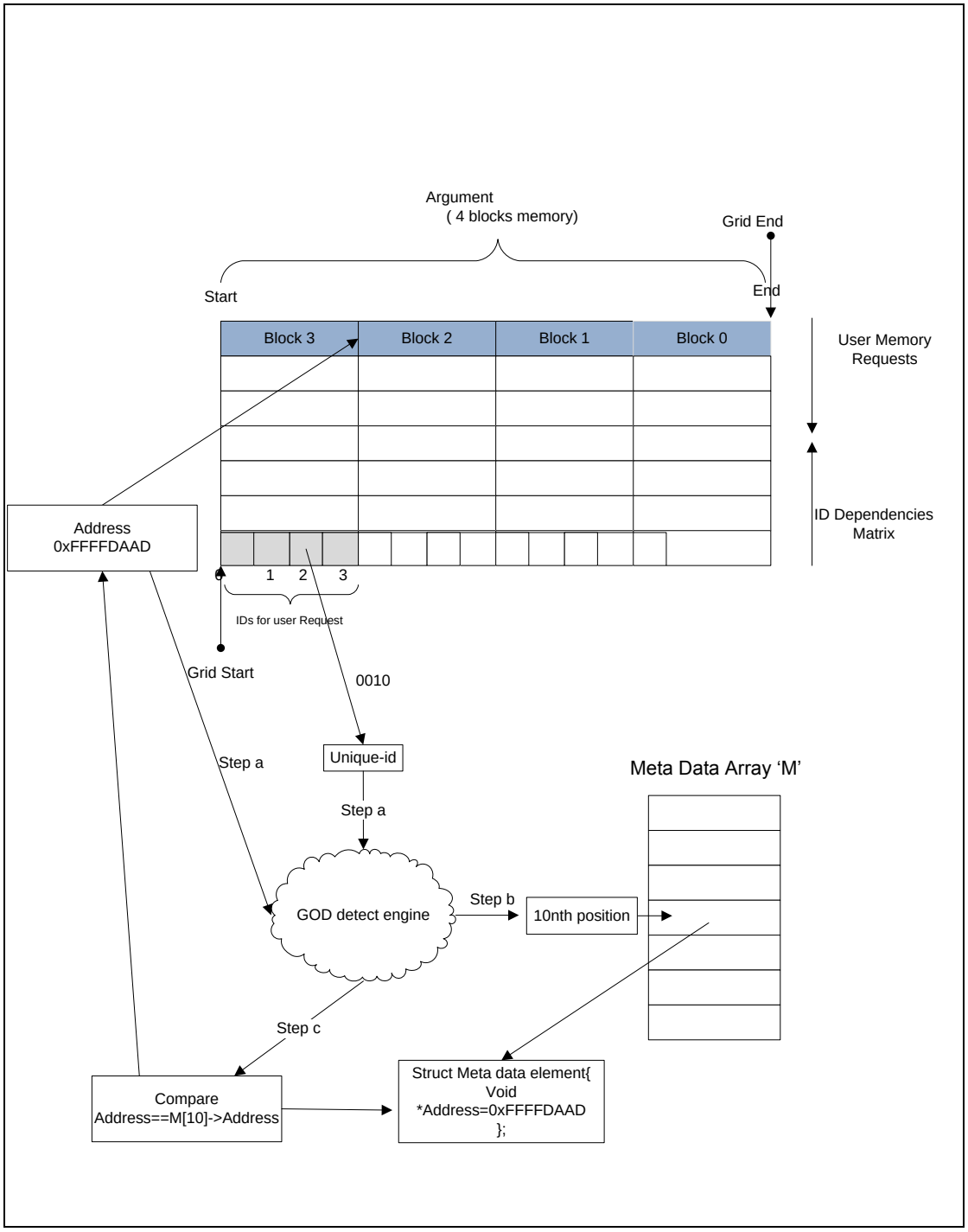

Figure 2.9—2 ADAM SMP Dependence detection. Example of the dependence detection operation using Grids.

Furthermore n the SMP version of ADAM because the workers and the Master share the address space it is possible to delegate part of the dependence analysis to the workers. The release of each executed task is performed by the workers in parallel with each other and with the master's issue.

3 Evaluation

We evaluate ADAM on a Playstation3 system equipped with 256MBs of RAM and a Cell Broadband Engine processor running at 3,192 GHz. Playstation3 systems allow for the use of, at most, 6 out of the 8 SPEs, therefore our experiments include a range of 1 to 6 workers and 1 Master.

In all measurements regarding ADAM, we include the following breakdowns for the Master and Worker.

PPE Breakdown

- **Instantiate**
  Instantiate is the time dedicated to creating a task. This time includes the overhead of the Dependence analysis required for each argument of the task.
- **Complete**
  Complete is the time required to update the dependency graph for each completed task.
- **Issue**
  Issue is the time required for the master to send the task descriptions to the workers. The task descriptor is send via remote stores.
- **Stalled**
  This is the time portion in which the Master has tasks eligible for execution and polls the worker queues for an available slot.
- **Wait**
  This is the measured time of the Master blocking until workers complete all issued tasks. This corresponds to the barrier synchronization time. ADAM requires only one barrier at the end of the application.
- **Writeback**
  Writeback is the measured time of the Master for copying renamed buffers back to the original addresses
- **PPC tasks time**
  This is the time the Master spends executing Tasks
- **Application**
  This is the time dedicated to the running application excluding the aforementioned overheads and initialization.

SPE Breakdown

- **Task Ticks**
  Task is the portion of time the SPE spends executing task code.
- **Lib Ticks**

Lib is the portion of time the SPE spends executing library code.

- **Idle Ticks**
  Idle is the time spent by an SPE when it has no pending or executing tasks.

For the evaluation of ADAM we use the following series of task-based benchmarks that originate from the SPLASH-2 benchmark suite, the CELLSS [**2**] runtime, and the Sequoia runtime [**4**]. The task window size in all the runs is set to 2048, and for all the evaluated applications we present runs that exceed the task window. The size of the worker queues is set to 2. Fine-tuning these parameters for each application can potentially yield better performance.

3.1 Applications

Each graph in this section represents runs that range from 1 to 6 workers. Each run allocates two bars in the graph, one with the PPE Breakdown of the master and one with the SPE breakdown of the average of all workers. The Y-axis is time in μ-seconds and the X-axis is number of SPEs.

3.1.1 LU

LU performs LU factorization on a non-contiguous matrix of blocks and it originates from the SPLASH-2 Benchmark suite. We created tasks that correspond to the following already defined kernels “bdiv”, “bmodd”, “bmod” and “lu0”, based on the LU port for the TPC runtime. We use two versions of this benchmark one with single precision numbers (labeled LUsp) and the original with double precision numbers (labeled LU). Each run is configured by two parameters, the size of the array, and the size of the blocks (this stands for application block size, not ADAM’s block size). The size of the array is the application dataset and allows us to control the number of produced tasks. We use two values for this parameter, either the default, 512, or 4096, which is the maximum dataset that fits into our memory. The size of the LU block affects the size of the task in terms of both data and execution as well as the number of tasks. The LU block size is configured with the following values 8,16,32. The number of produced tasks for each configuration can be seen in **Table 3.1.1—I:**

Runs	Number of tasks
512x512(8x8)	89.440
512x512(16x16)	11.440

512x512(32x32)	1.496
4096x4096(8x8)	44.870.400
4096x4096(16x16)	5.625.216
4096x4096(32x32)	707.264

Table 3.1.1—I Number of tasks for the LU application

The complexity of the dependencies for the LU benchmark remains unaffected by the number of tasks. The number of tasks affects only the size of the dependencies graph. The number of tasks in this application does not affect ADAMs behavior in terms of scalability, an argument validated when examining runs with the same block size but different array sizes. The key aspect that affects scalability is the size of the tasks. As we increase the application's block size ADAMs scalability improves. Furthermore by comparing runs with the same configuration but with different precision numbers (LU to LUsp), we see that runs with double precision always scale better. Due to the fact that ADAM is a runtime system, ADAM introduces overhead to each run. When the overhead exceeds the SPE task time plus the SPE lib time then ADAM's overhead becomes the bottleneck that limits scalability. For each run the following condition should be respected:

Instantiate + Complete + Issue + Writeback ≤ Task + Lib

Respecting this condition will ensure that ADAM will not impair the applications scalability but it does not guarantee that the running application will scale. Task and Lib correspond to the average of all the participating workers.

In the runs displayed in **Figure 3.1.1—1** and **Figure 3.1.1—4** ADAM fails to scale with runs with more than 2 SPEs, due to the introduced overhead. While these two sets of runs are for different datasets the task size remains the same. The task size in these two sets of runs corresponds for 8x8 blocks(Application blocks). Scalability improves as the task size increases(**Figure 3.1.1—2** and **Figure 3.1.1—3**) for the 512x512 dataset as well as for the 4096x4096 dataset(**Figure 3.1.1—5** and **Figure 3.1.1—6**).

In the runs concerning the single precision version of LU, the runs are equivalent to the double precision runs in terms of dependencies. The size of the tasks however is smaller both in terms of data and execution. ADAM fails to scale due to overhead in **Figure 3.1.1—7** and **Figure 3.1.1—10** where the block size is 8x8(Application block). ADAMs also fails to scale due to overhead in the runs with 16x16 block size(Application block) in **Figure 3.1.1—8** and **Figure 3.1.1—11** although the introduced overhead is identical to the double precision runs

in **Figure 3.1.1—2** and **Figure 3.1.1—5**. The reason is that the task-size in the two single precision LU sets of runs is smaller than the corresponding double precision runs.

The absence of excessive writeback overhead despite the fact that the number of tasks(**Table 3.1.1—I**) exceeds the size of the task in all the runs is because ADAM successfully identifies intermediate values. Furthermore it indicates that the size of the Task window does not impact scalability in a negative way in any of these runs.

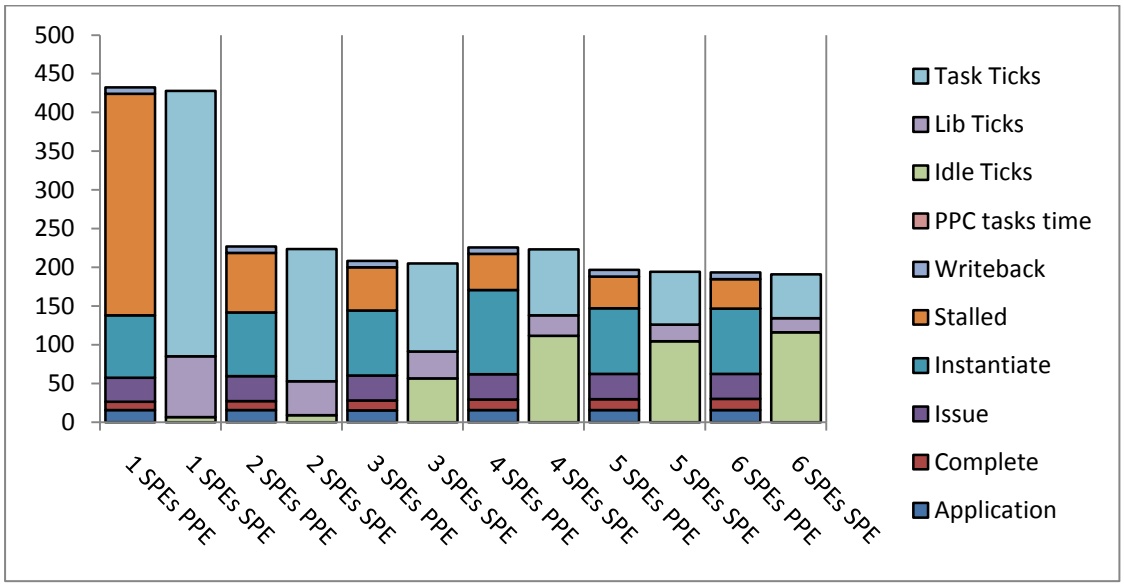

Figure 3.1.1—1 LU 512x512(8x8). LU execution Breakdown for the PPE and the SPEs for runs with 1,2,3,4,5,6 SPEs, for an array size of 512x512 with application block size 8x8.

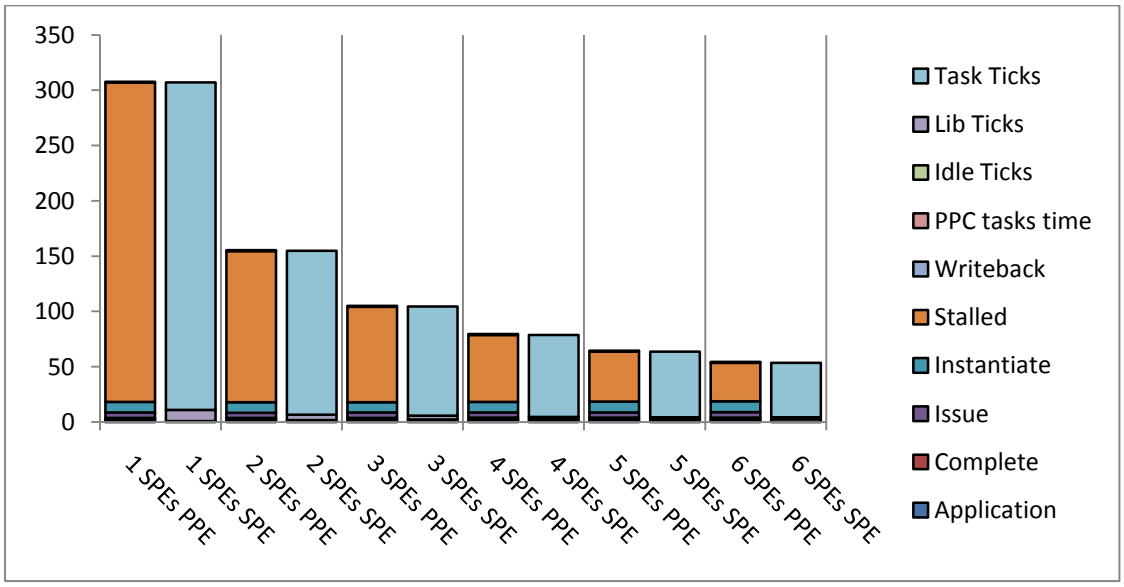

Figure 3.1.1—2 LU 512x512(16x16).LU execution Breakdown for the PPE and the SPEs for runs with 1,2,3,4,5,6 SPEs, for an array size of 512x512 with application block size 16x16.

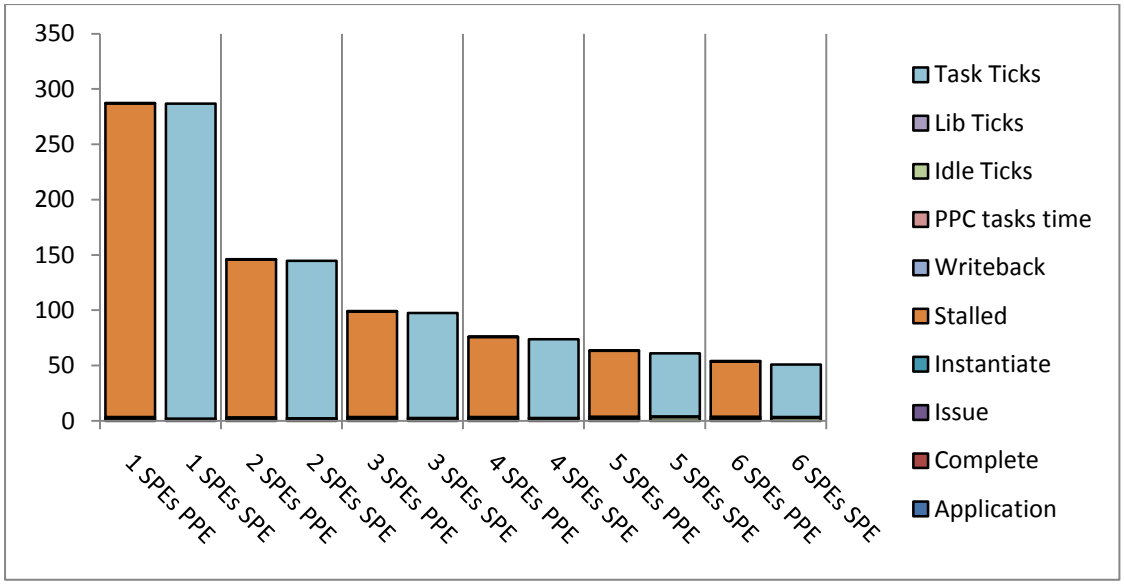

Figure 3.1.1—3 LU 512x512(32x32).LU execution Breakdown for the PPE and the SPEs for runs with 1,2,3,4,5,6 SPEs, for an array size of 512x512 with application block size 32x32.

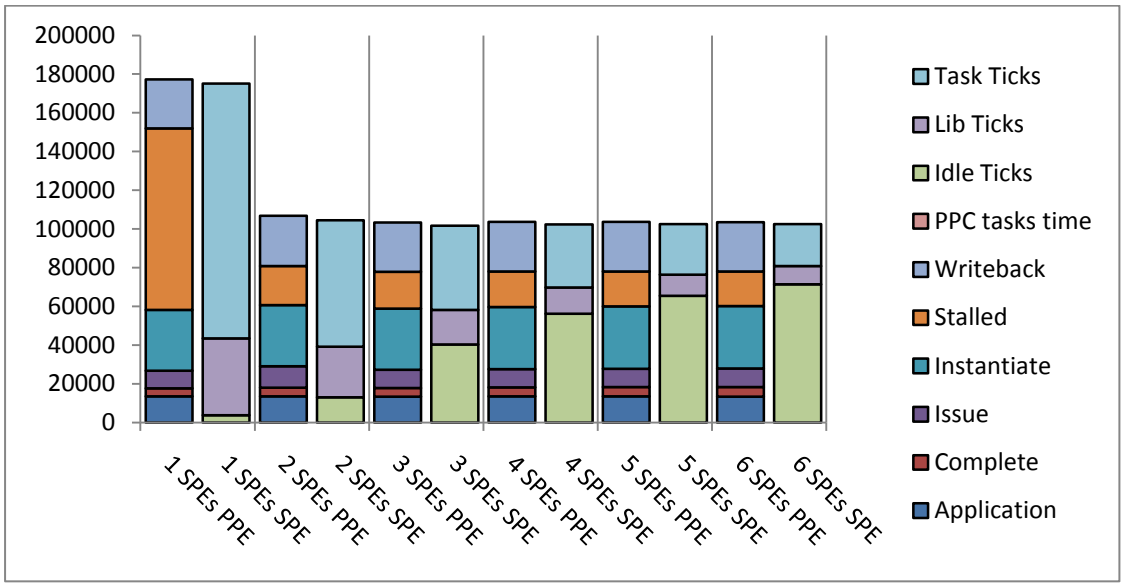

Figure 3.1.1—4 LU 4096x4096(8x8). LU execution Breakdown for the PPE and the SPEs for runs with 1,2,3,4,5,6 SPEs, for an array size of 4096x4096 with application block size 8x8.

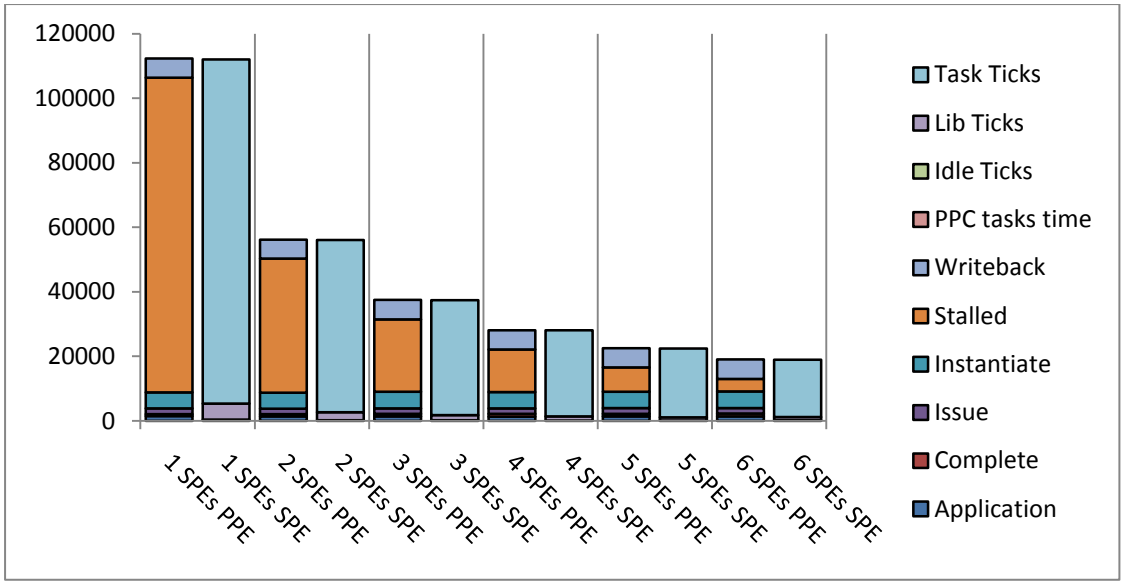

Figure 3.1.1—5 LU 4096x4096(16x16).LU execution Breakdown for the PPE and the SPEs for runs with 1,2,3,4,5,6 SPEs, for an array size of 4096x4096 with application block size 16x16.

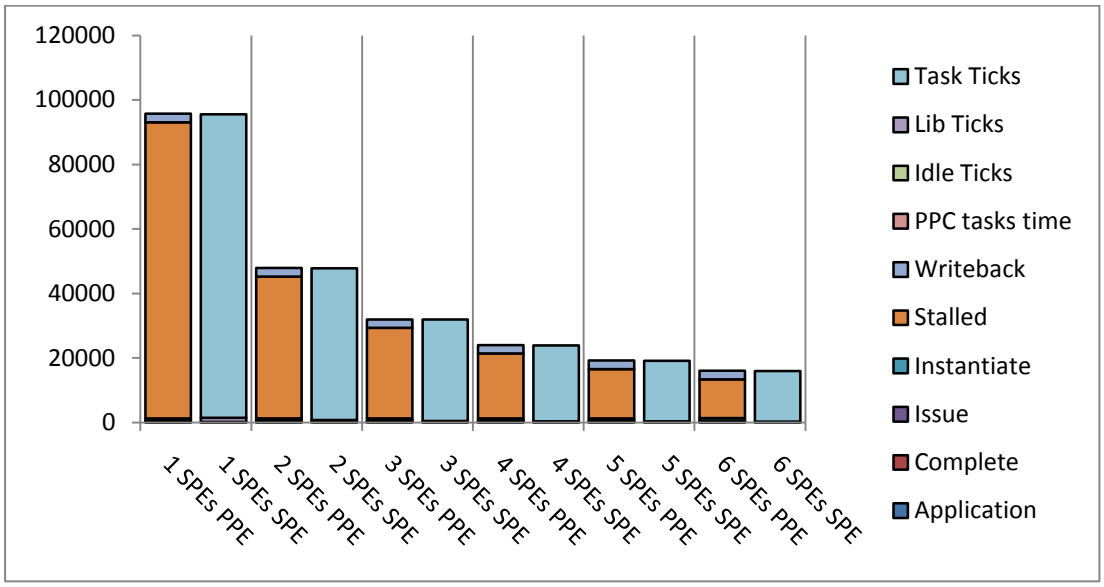

Figure 3.1.1—6 LU 4096x4096(32x32).LU execution Breakdown for the PPE and the SPEs for runs with 1,2,3,4,5,6 SPEs, for an array size of 4096x4096 with application block size 32x32.

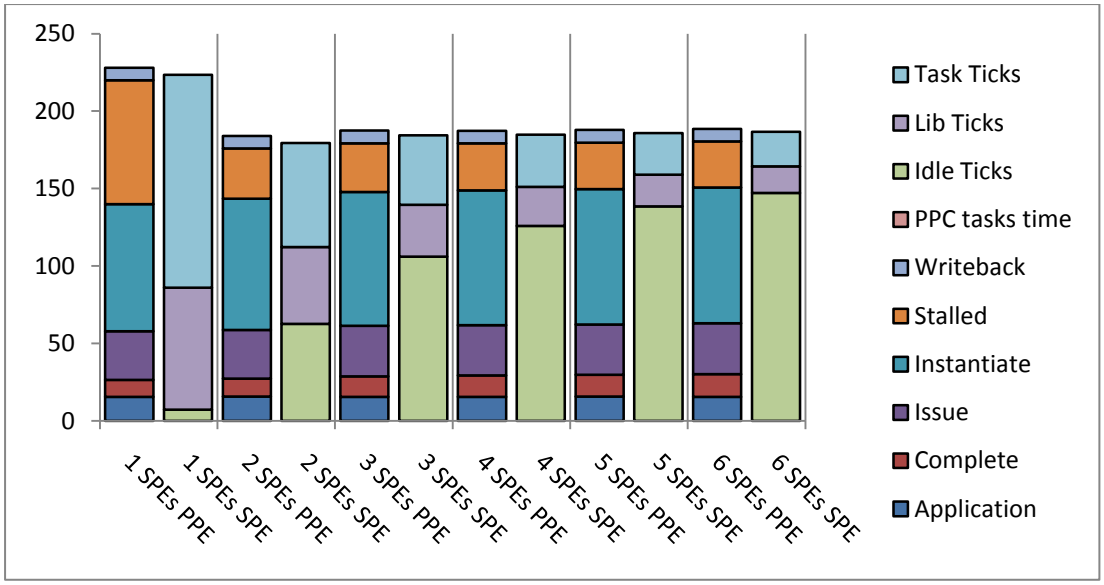

Figure 3.1.1—7 LUsp 512x512(8x8). LU with single precision, execution Breakdown for the PPE and the SPEs for runs with 1,2,3,4,5,6 SPEs, for an array size of 512x512 with application block size 8x8.

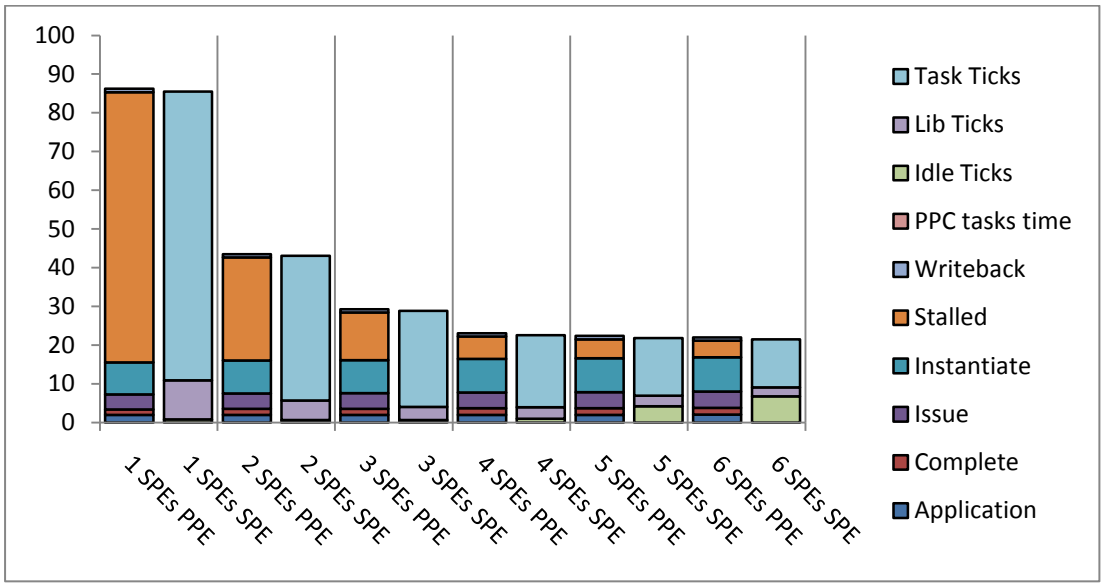

Figure 3.1.1—8 LUsp 512x512(16x16). LU with single precision, execution Breakdown for the PPE and the SPEs for runs with 1,2,3,4,5,6 SPEs, for an array size of 512x512 with application block size 16x16.

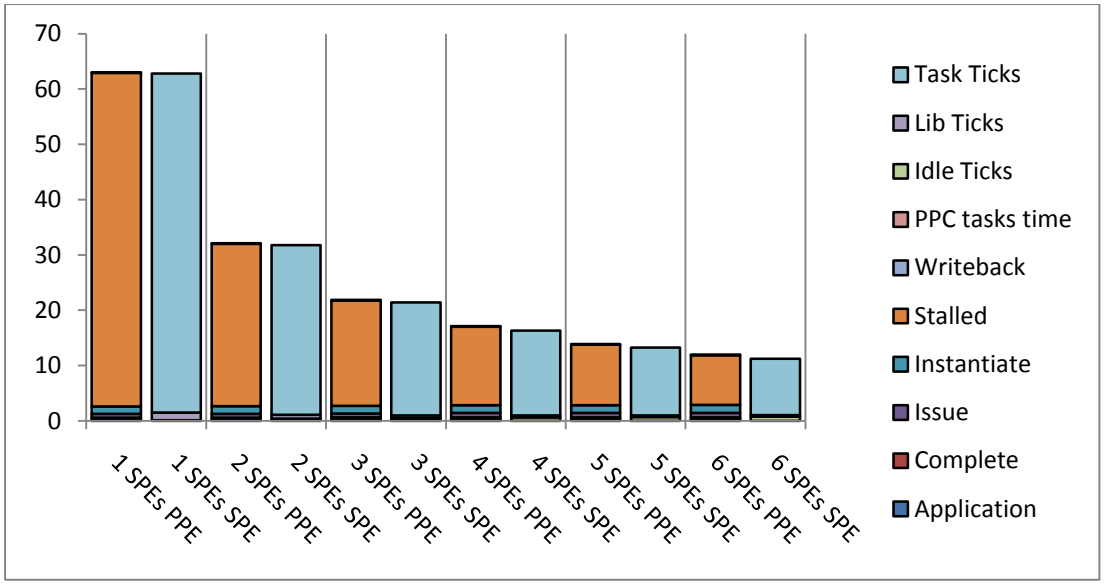

Figure 3.1.1—9 LUsp 512x512(32x32). LU with single precision, execution Breakdown for the PPE and the SPEs for runs with 1,2,3,4,5,6 SPEs, for an array size of 512x512 with application block size 32x32.

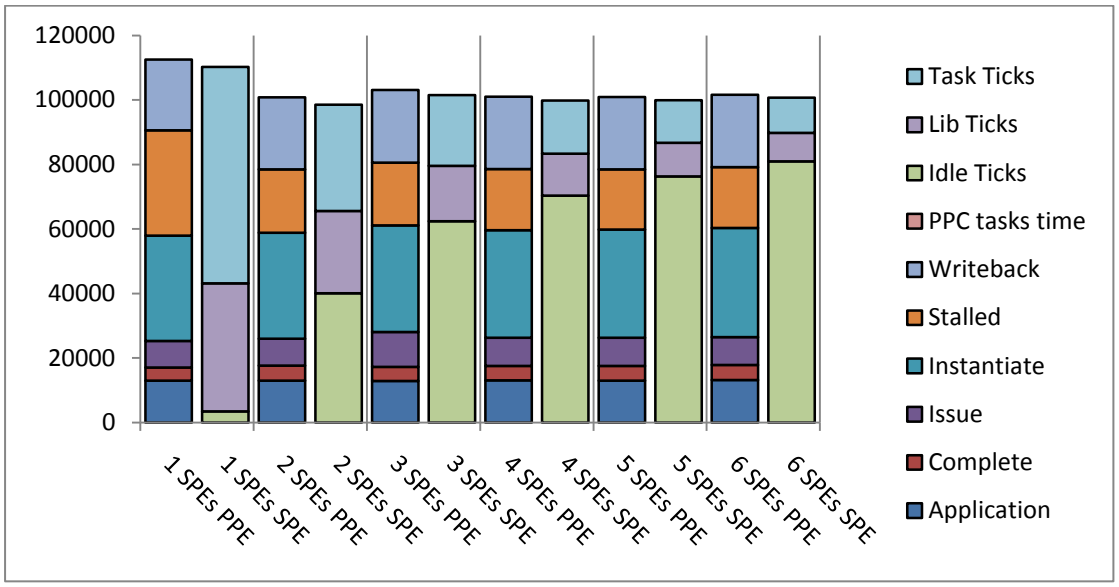

Figure 3.1.1—10 LUsp 4096x4096(8x8). LU with single precision, execution Breakdown for the PPE and the SPEs for runs with 1,2,3,4,5,6 SPEs, for an array size of 4096x4096 with application block size 8x8.

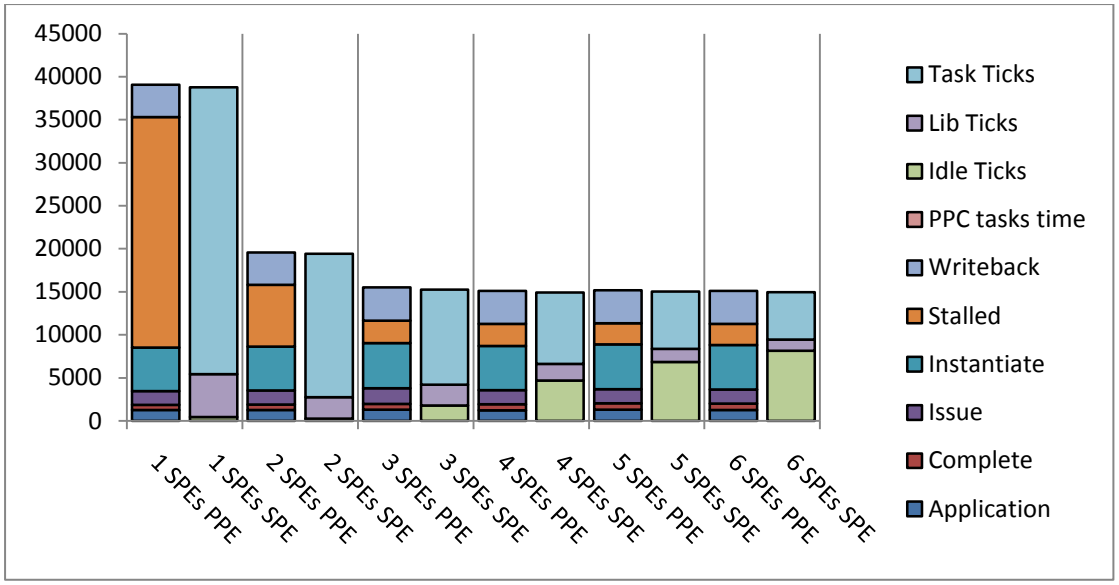

Figure 3.1.1—11 LUsp 4096x4096(16x16). LU with single precision, execution Breakdown for the PPE and the SPEs for runs with 1,2,3,4,5,6 SPEs, for an array size of 4096x4096 with application block size 16x16.

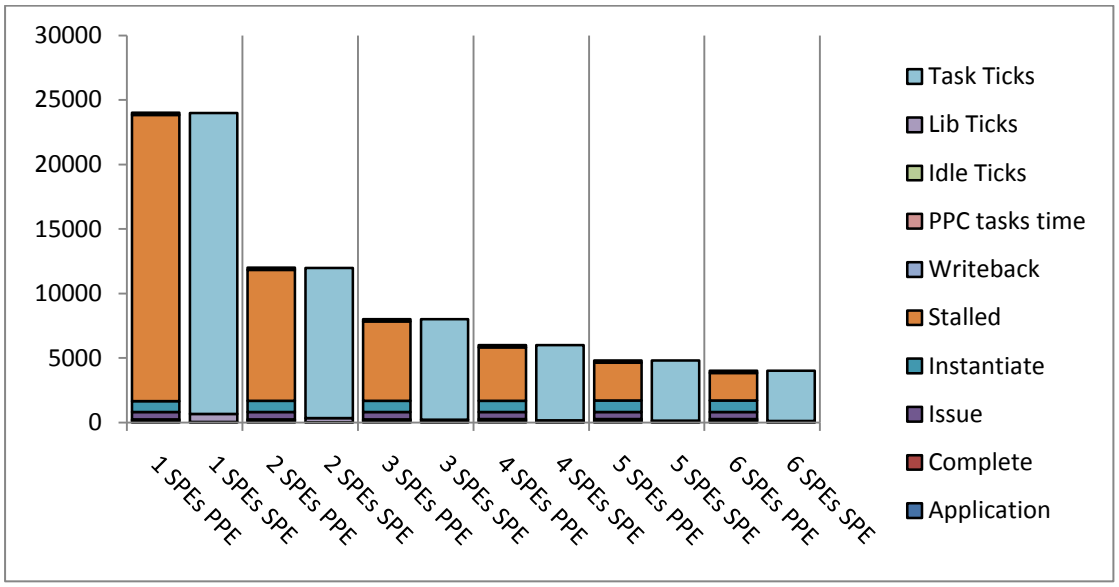

Figure 3.1.1—12 LUsp 4096x4096(32x32). LU with single precision, execution Breakdown for the PPE and the SPEs for runs with 1,2,3,4,5,6 SPEs, for an array size of 4096x4096 with application block size 32x32.

3.1.2 FFT

FFT performs a two dimensional Fast Fourier Transformation and originates also from the SPLASH-2 Benchmark suite. We use the task-based version originally created for the TPC runtime that offloads the FFT steps and the matrix transposition to the workers. Similar to LU we used two versions of this benchmark, one with single precision numbers and the original with double precision numbers. We perform two runs with each version, one with

64 thousand elements and one with 4 million elements. **Table 3.1.2—I** shows the number of tasks produced for each configuration.

Runs	Number of tasks
FFT(64k)	572
FFT(4M)	20.672

Table 3.1.2—I Number of tasks for the FFT application

This application performs blocked transposition delegated to the workers via tasks. In order to express blocks as task arguments we use strided arguments. Each element of a strided argument is translated to an argument internally and treated as such, independently by ADAM. For the case of the FFT when a task is issued with 3 strided arguments of 32 elements each, ADAM treats it as a task with 96 arguments. Although the amount of analysis required is disproportionally large in relation to the number of tasks the application scales well in the case of double precision numbers. In runs with single precision the overheads dominate the execution time. We also observed a fair amount of writeback time introduced. The writeback overhead relates to the excessive number of total meta-data elements required to handle all these arguments compared to the size of the task window, and if we compare single to double precision runs we see that it remains constant regardless the size of the tasks.

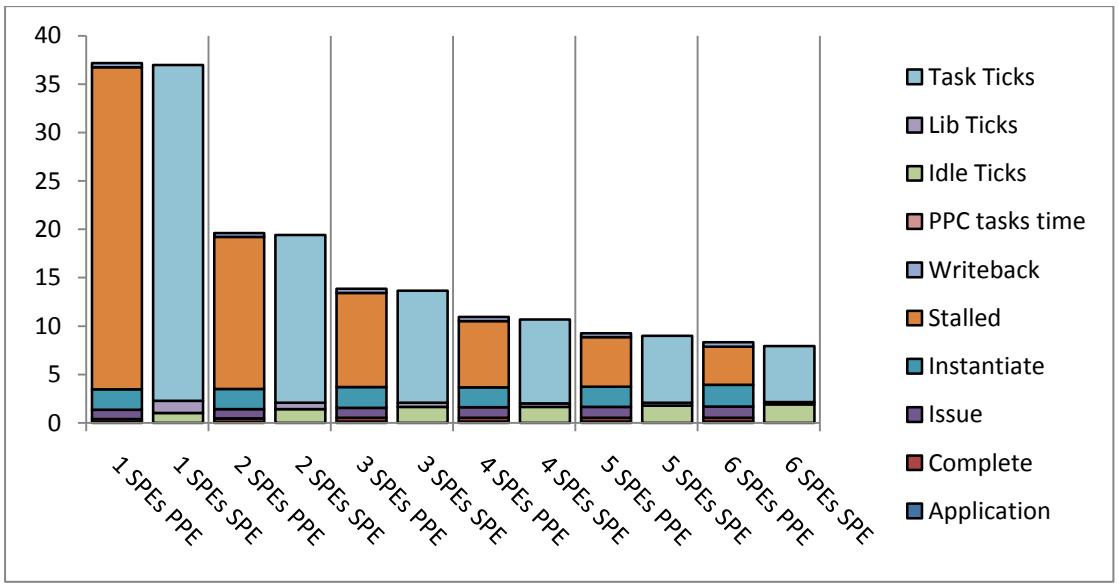

Figure 3.1.2—1 FFT 64k. FFT execution Breakdown for the PPE and the SPEs for runs with 1,2,3,4,5,6 SPEs, for 16384 Complex Double elements.

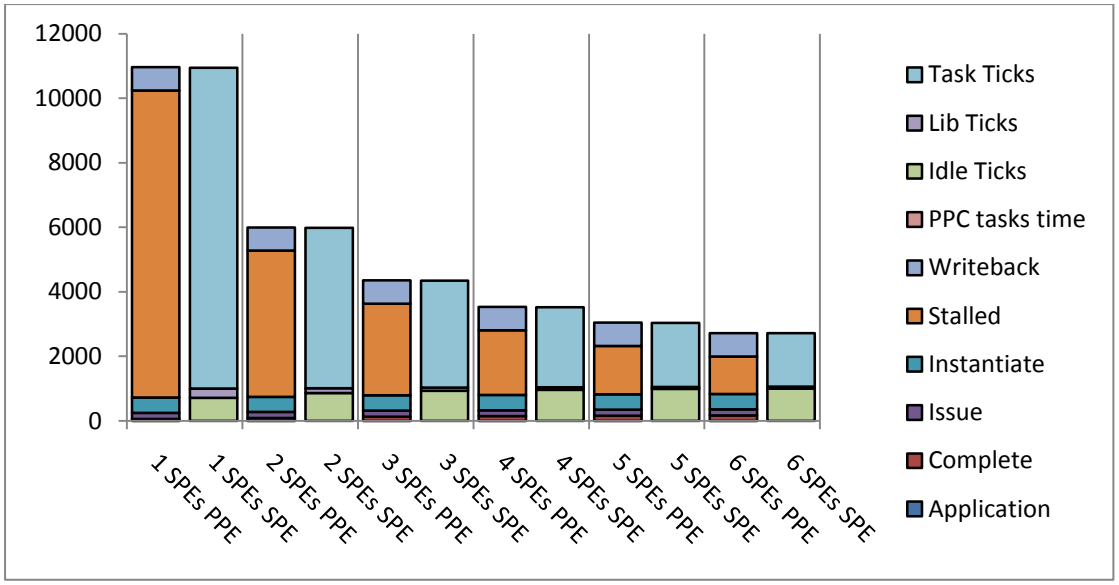

Figure 3.1.2—2 FFT 4M. FFT execution Breakdown for the PPE and the SPEs for runs with 1,2,3,4,5,6 SPEs, for 4194304 Complex Double elements.

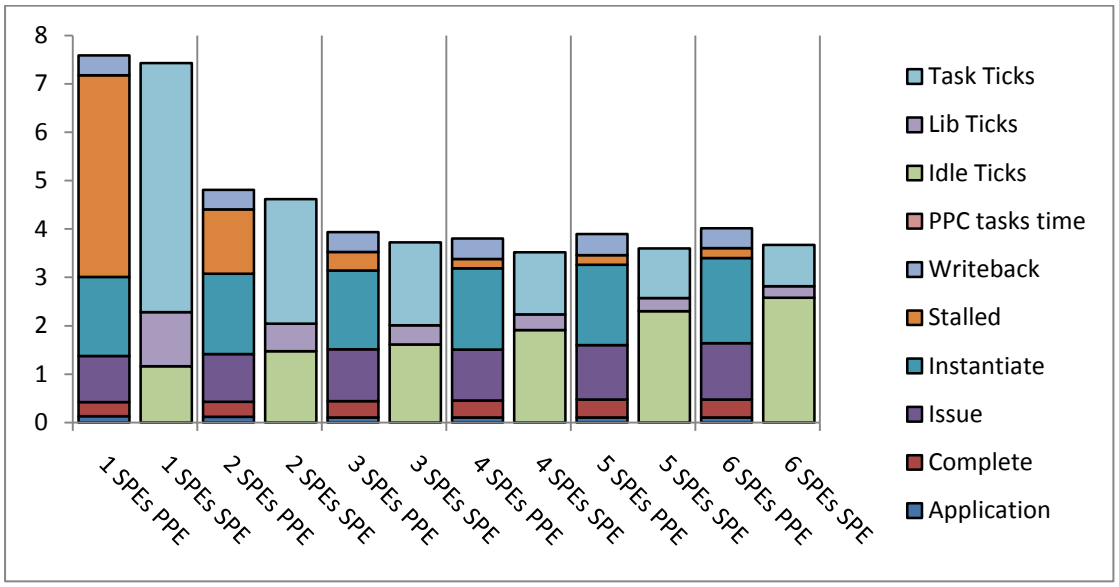

Figure 3.1.2—3 FFTsp 64k. FFT with single precision, execution Breakdown for the PPE and the SPEs for runs with 1,2,3,4,5,6 SPEs, for 16384 Complex Float elements.

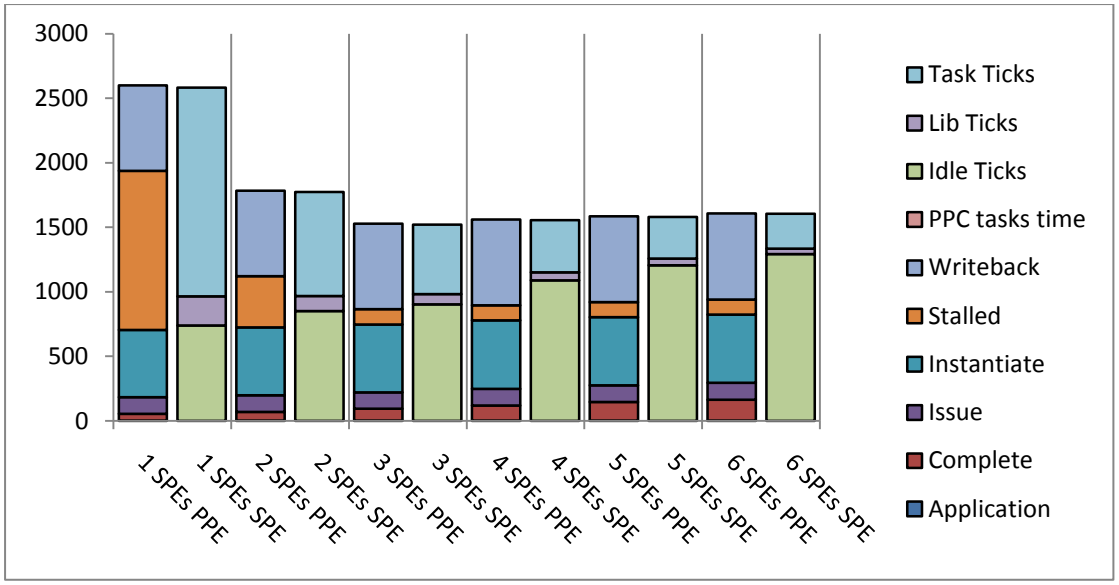

Figure 3.1.2—4 FFTsp 4M. FFT with single precision numbers, execution Breakdown for the PPE and the SPEs for runs with 1,2,3,4,5,6 SPEs, for 4194304 Complex Float elements.

3.1.3 Sequoia [4] Kernels

We use two Sequoia [**4**] kernels originally ported for the TPC runtime called saxpy and sgemv. Saxpy and sgemv are both communication bound. Saxpy contains of one kernel that consists of a vectorized multiply and add between a block and an alpha value(*y= y + alpha*x*). Sgemv is a vectorized MxN matrix multiplication and add between two blocks, an alpha value and a beta value (*y= beta*y + alpha*A*x*). These kernels consist of tasks that are independent with each other. For statistical validity each experiment is internally executed 100 times and blocked by a barrier after each iteration. In ADAM we remove this barrier and allow for the 100 iterations of the experiment to occur as one unified experiment. This creates dependencies among tasks because each iteration of the experiment uses the same data. The task window size is to 2048 and because it is large enough for ADAM to contain tasks from 2 consecutive iterations, writeback overhead is minimal. **Table 3.1.3—I** shows the number of produced tasks for the saxpy benchmark. The saxpy block size is 1024. The behavior of the runtime remains the same in saxpy regardless the block size, and thus we only include runs with 1024 block size in our evaluation.

Runs	Number of tasks
Saxpy	102.400

Table 3.1.3—I Number of tasks for the Saxpy benchmark

ADAM does not scale well for the Saxpy becnhmark, because the runtime-introduced overheads dominate the execution time. Although Saxpy does not scale anyway

in **Figure 3.1.3—1** this does not occur due to the fact that the application is communication bound but rather because the duration of the tasks(Lib+Task) is less than the introduced overheads.

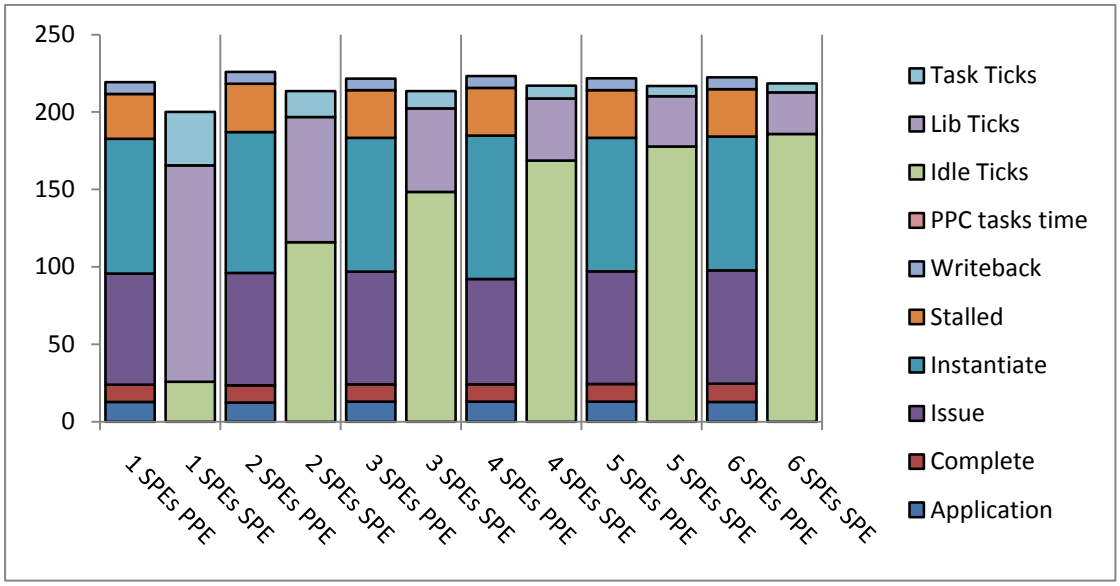

Figure 3.1.3—1 SAXPY. Saxpy execution Breakdown for the PPE and the SPEs for runs with 1,2,3,4,5,6 SPEs.

Sgemv is a computation bound benchmark. We perform two runs for this application with the N parameter set as 4 and 8 respectively. In (*y= beta*y + alpha*A*x*) x and y are of size M where M is set to 1024 while A is of size Mx(N*1024). The N parameter affects the size of the task in terms of both data transfer size and execution. **Table 3.1.3—II** displays the number of produced tasks for each run:

Runs	Number of tasks
SGEMV N4	10.240
SGEMV N8	10.240

Table 3.1.3—II Number of tasks for the Sgemv benchmark

In **figures Figure 3.1.3**—2 and **Figure 3.1.1**—3 we see that as the task size increases from the run with N4 to the run with N8, scalability improves. Although SGEMV does not scale anyway, like in the case of saxpy the cause here is the overhead introduced by ADAM and not the application.

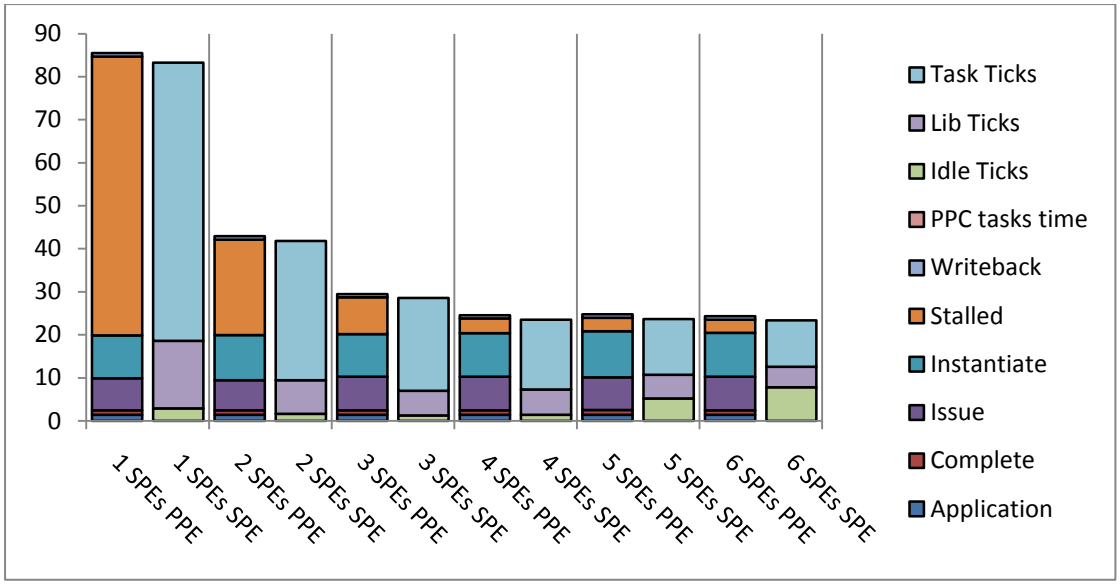

Figure 3.1.3—2 SGEMV N4. Sgemv execution Breakdown for the PPE and the SPEs for runs with 1,2,3,4,5,6 SPEs and N=4.

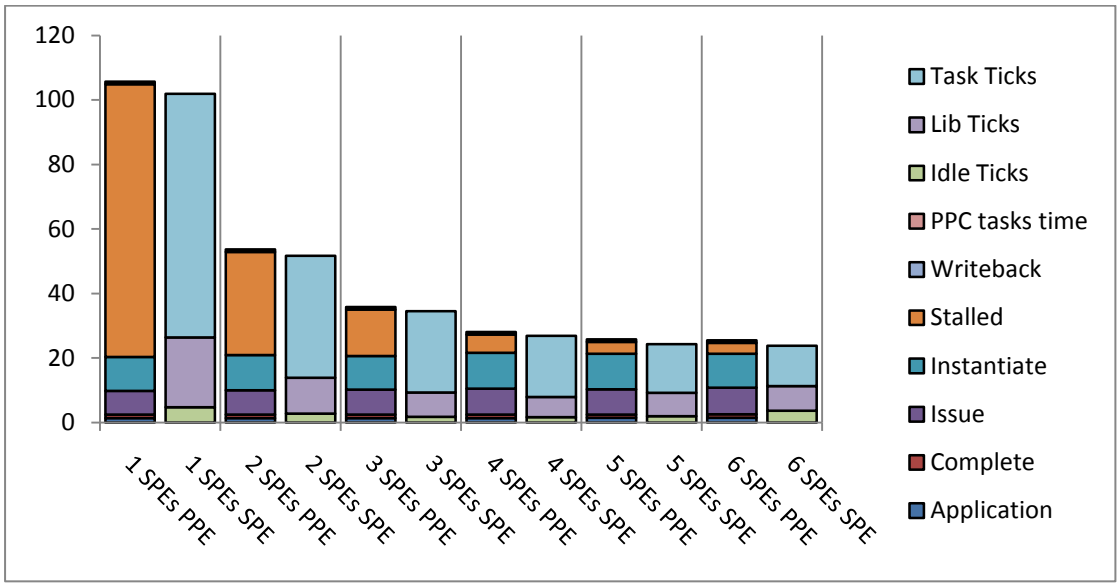

Figure 3.1.3—3 SGEMV N8. Sgemv execution Breakdown for the PPE and the SPEs for runs with 1,2,3,4,5,6 SPEs and N=8.

3.1.4 Cholesky

This is a benchmark from the CELLSS [**2**] runtime. It performs Cholesky factorization on a sparse matrix of blocks. We run this benchmark with two datasets one that consists of a 13x13 matrix of blocks and one that consists of a 20x20 matrix of blocks. Each block is a 64x64 matrix of floats. **Table 3.1.4—I** shows the number of produced tasks for each run. Cholesky is a computation bound benchmark with a highly complex dependency graph. Because the introduced overhead is low compared to the application task time, and despite

the complexity of the dependency graph, the application exhibits good scalability as seen in **Figure 3.1.4—1** and **Figure 3.1.4—2.** Cholesky is an application that exhibits a lot of dependencies compared to the number of produced tasks and therefore requires either a lot of synchronization either a lot of dependence analysis. Because ADAM provides the benefits of dependence analysis at a relatively low cost, Cholesky scales well.

Runs	Number of tasks
cholesky (13x13)	455
cholesky(20x20)	1540

Table 3.1.4—I Number of tasks for the Cholesky benchmark

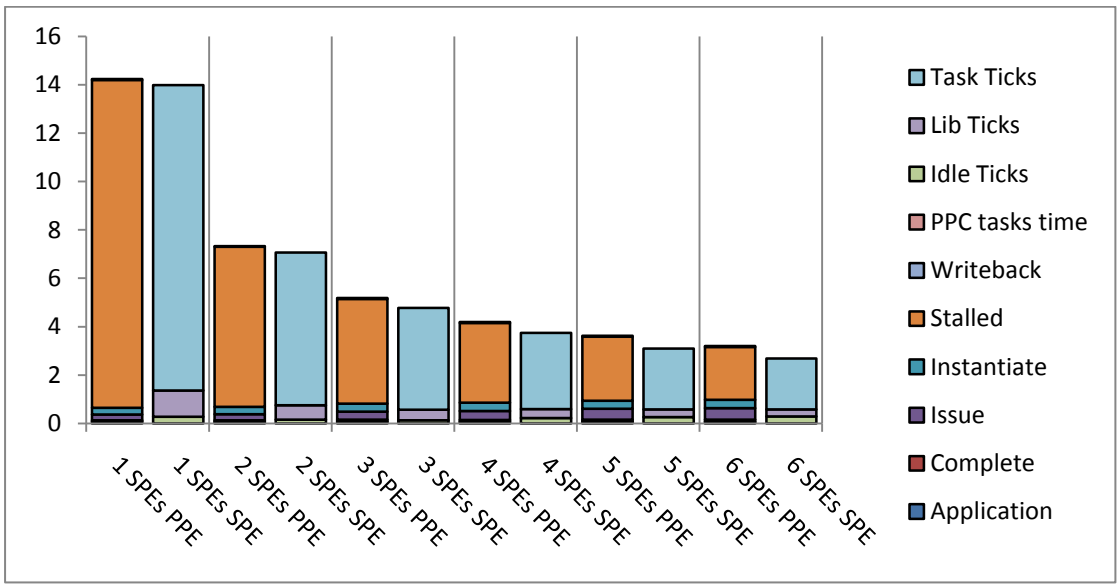

Figure 3.1.4—1 Cholesky 13x13. Cholesky execution Breakdown for the PPE and the SPEs for runs with 1,2,3,4,5,6 SPEs and for a 13x13 matrix of 64x64 blocks(Cholesky blocks) of single precision numbers.

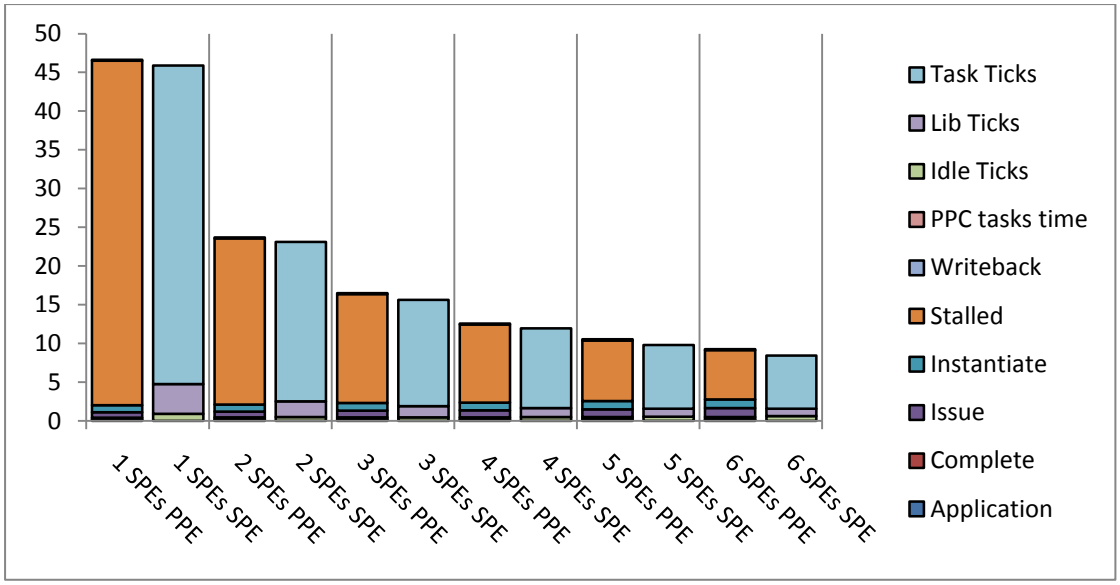

Figure 3.1.4—2 Cholesky 20x20. Cholesky execution Breakdown for the PPE and the SPEs for runs with 1,2,3,4,5,6 SPEs and for a 20x20 matrix of 64x64 blocks(Cholesky blocks) of single precision numbers.

3.1.5 Matmul

This is a matrix multiplication benchmark from the CELLSS [**2**] runtime. The Tasks in this benchmark are created through 3 nested loops with each task of innermost loop dependent on the preceding task, while each task is independent from tasks from different iterations of the two outermost loops. For our evaluation purposes we use the non-vectorized version of this benchmark. We run this benchmark for two datasets one that consists of a 13x13 matrix of blocks and one that consists of a 20x20 matrix of blocks. Each block is a 64x64 matrix of floats. **Table 3.1.5—I** shows the number of produced tasks for each run:

Runs	Number of tasks
Matmul 13x13	2.197
Matmul 20x20	8.000

Table 3.1.5—I Number of tasks for the Matmul benchmark

This benchmark is an example of how ADAM can export parallelization from seemingly sequential applications. Since each task of the innermost loop is dependent on the one previously issued, tasks of the same iteration should execute in sequence. Tasks from different iterations, of the two outermost loops, however can execute in parallel. ADAM through dependence analysis reorders tasks and takes advantage of this parallelism. The reordering of tasks can been seen in **Figure 3.1.5—1**.

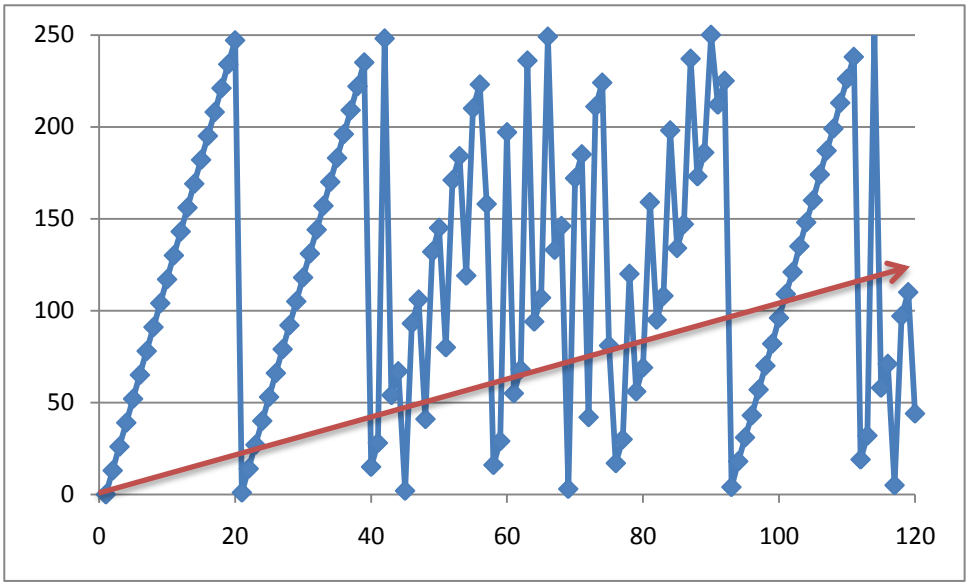

Figure 3.1.5—1 Task re-ordering. Ordering of the 120 tasks of a matmul 13x13 run. X-axis is execution order while y-axis is the issue order.

This graph represents the order of the first 120 tasks of a 13x13 Matmul execution. The innermost loop in this run consists of 13 iterations. Each task in this graph is represented by an (x,y) point where x is an id representing the order that ADAM executes tasks(more precisely assigns them to workers) and y is an id representing the order, this task was issued to ADAM (program order). The red line represents the sequential execution of the program where for each task, the issue order would coincide with the execute order, hence x=y. Tasks above the red line are tasks executing ahead of time (future tasks) while tasks below the red line are tasks executing late(past tasks). **Figure 3.1.5—2** shows the reordering of the tasks for the entire run.

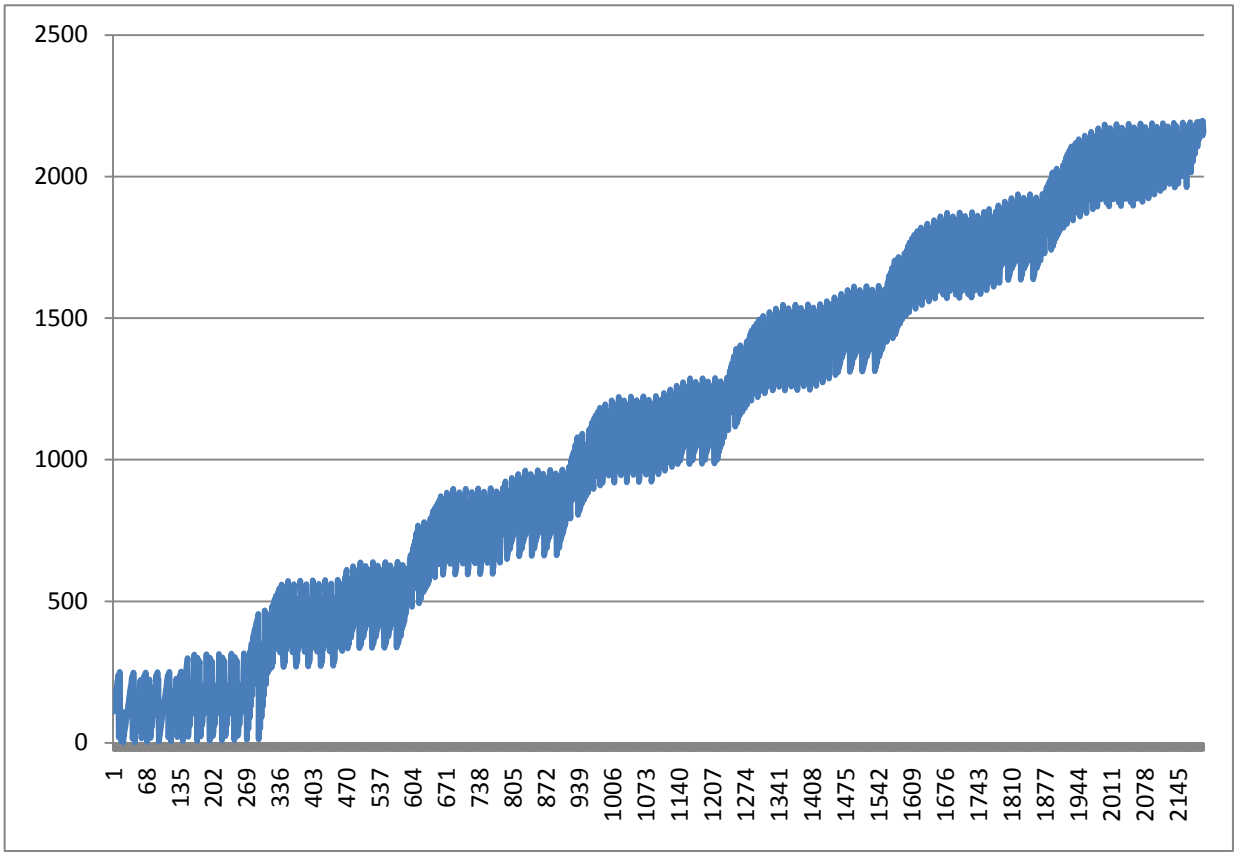

Figure 3.1.5—2 Task re-ordering. Ordering of all(2145) of the tasks of a matmul 13x13 run. X-axis is execution order while y-axis is the issue order.

We see in **Figure 3.1.5—3** and **Figure 3.1.5—4** that this application scales well due to the exported parallelism from ADAM and the large size of the tasks in terms of computation.

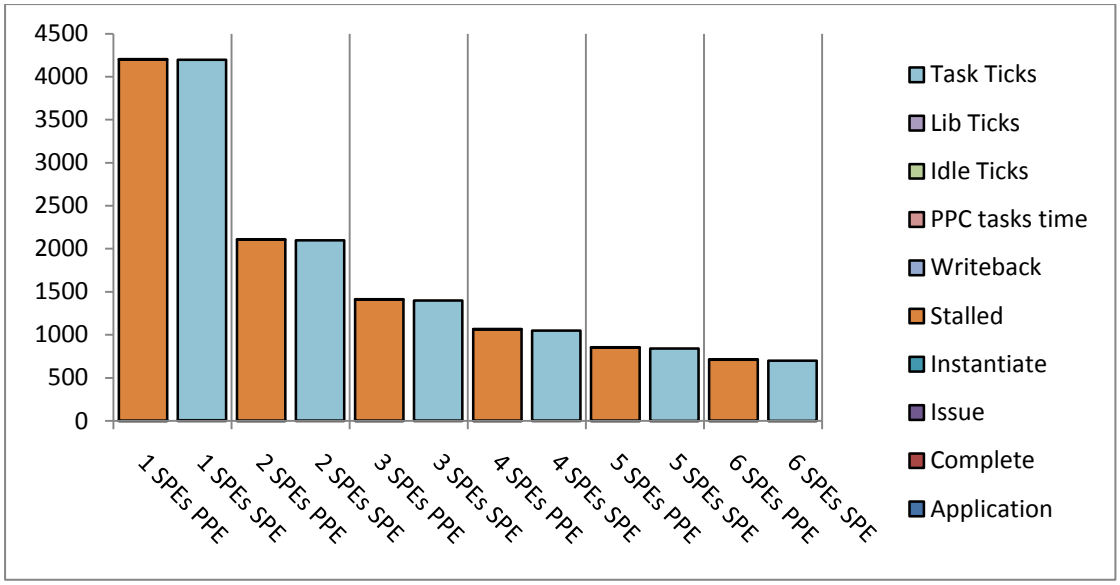

Figure 3.1.5—3 Matmul 13x13. Matmul execution Breakdown for the PPE and the SPEs for runs with 1,2,3,4,5,6 SPEs and for a 13x13 matrix of 64x64 blocks(Matmul blocks) of single precision numbers.

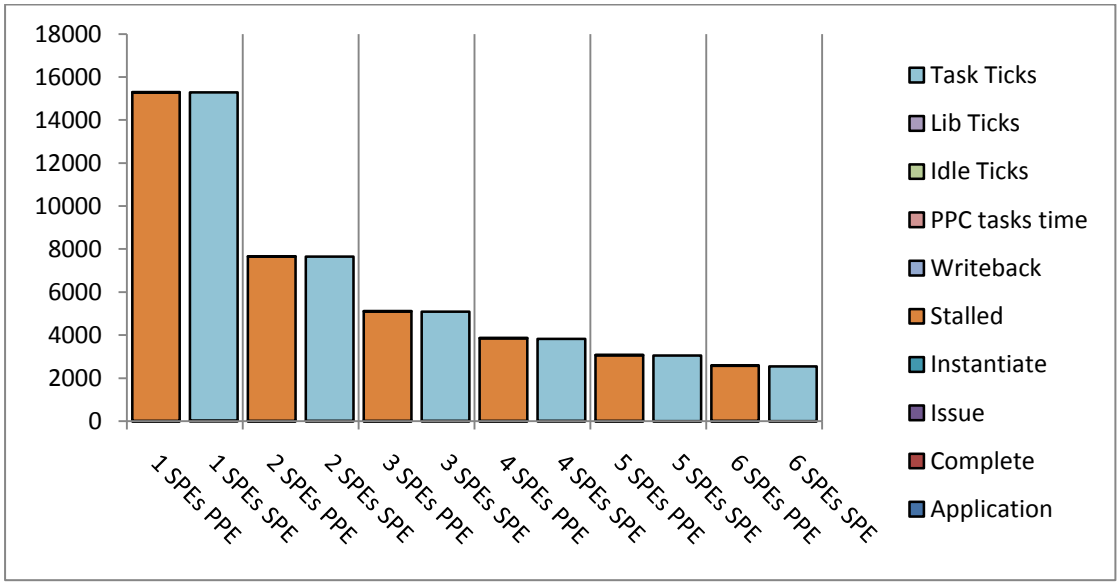

Figure 3.1.5—4 Matmul 20x20. Matmul execution Breakdown for the PPE and the SPEs for runs with 1,2,3,4,5,6 SPEs and for a 20x20 matrix of 64x64 blocks(Matmul blocks) of single precision numbers.

3.1.6 Jacobi

Jacobi is also a benchmark from the CELLSS [**2**] runtime. The dataset of this benchmark consists of a 32x32 matrix of blocks with each block consisting of 32x32 floats. We run this benchmark for two configurations one for 13 iterations of the Jacobi calculation and one for 20. We iterate a sparse matrix and perform the Jacobi method for each block using its neighboring blocks. The neighboring blocks are copied via tasks into four blocks called lefthalo, righthalo, bottomhalo and tophalo, which are subsequently used for the calculation of each block. Because for each block of the matrix we use the same four blocks to store the neighboring blocks and we copy the desired blocks on them, the calculation for each iterated block is dependent on the calculation of the previous iteration. **Table 3.1.6—I** contains the number of produced tasks for each run:

Runs	Number of tasks
Jacobi 13	66.560
Jacobi 20	102.400

Table 3.1.6—I Number of tasks for the Jacobi benchmark

This benchmark is a prime example of the effectiveness of data-renaming. For the Jacobi calculation of each block we use the same variables to store the neighboring blocks, therefore each calculation is dependent on the previous one. Expressed this way this program is seemingly sequential. Renaming however resolves the dependencies regarding these variables, exporting parallelism among the calculations of the blocks. The reason this benchmark does not scale is because the tasks are small and extremely unbalanced, since for every 5 issued tasks we have 4 small tasks that fill the 4 aforementioned variables with the neighboring blocks.

As we can see in **Figure 3.1.6—1** and **Figure 3.1.6—2** Jacobi fails to scale due to the introduced overhead. The task sizes are extremely small compared to the introduced overhead. The dominant overhead is the instantiate part, which increases, as the number of workers increases. This is because the number of Blocking Dependencies(**3.2 Parameters and Features**) increases as the number of workers increases. The reason for this is that ADAM with more workers stalls less and therefore can create more tasks, hence therefore more blocking dependencies appear.

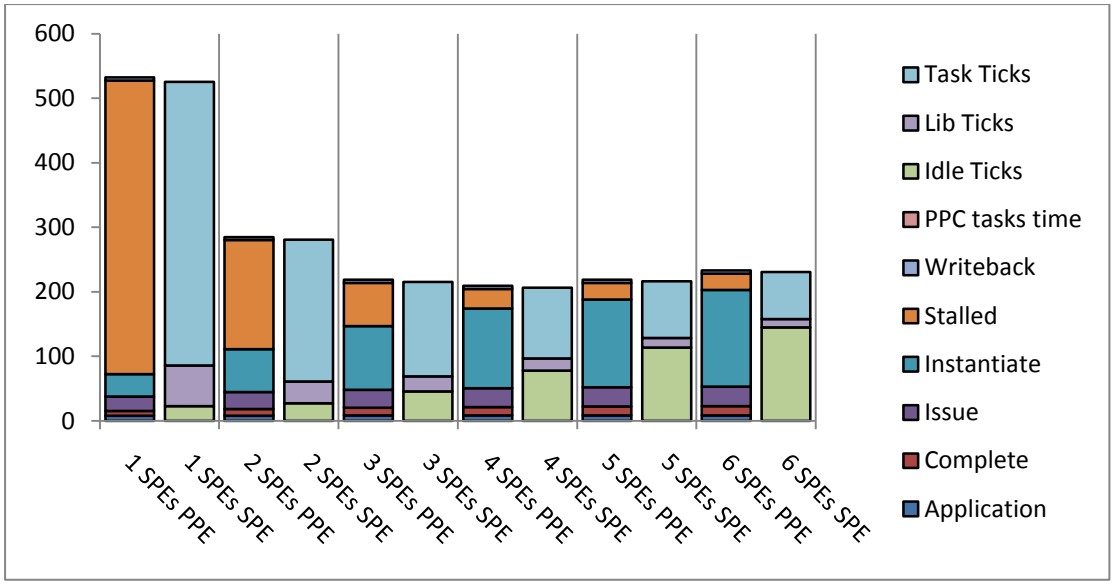

Figure 3.1.6—1 Jacobi 13. Jacobi execution Breakdown for the PPE and the SPEs for runs with 1,2,3,4,5,6 SPEs and for 13 iterations on a 32x32 matrix of 32x32 blocks(Matmul blocks) of single precision numbers.

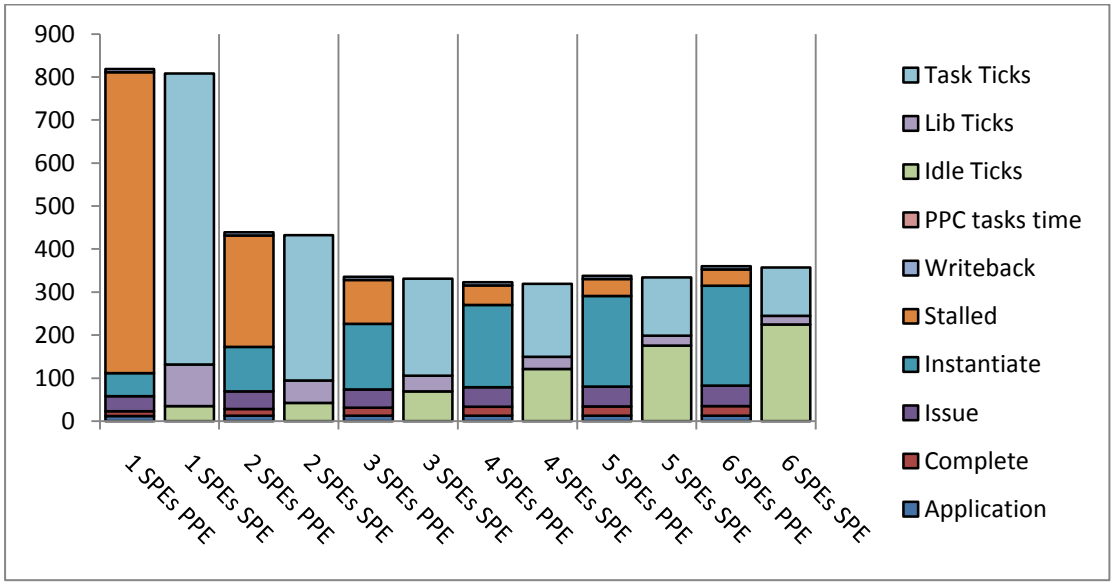

Figure 3.1.6—2 Jacobi 20. Jacobi execution Breakdown for the PPE and the SPEs for runs with 1,2,3,4,5,6 SPEs and for 20 iterations on a 32x32 matrix of 32x32 blocks(Matmul blocks) of single precision numbers.

3.2 Parameters and Features

ADAMs as previously described has a set of parameters and features that affect performance. In this section we examine their individual impact on ADAMs behavior. The parameters we will examine are the Task window, the dependency block size and the feature of running tasks on the PPE in addition to running tasks on the SPEs of the Cell processor. As a test case we use the double precision LU benchmark configured with data set size 512x512 and application block size 16x16. The total number of tasks produced in this configuration is 11440. Below we show an example dependency graph for the LU application. For reasons of readability this graph is actually from a smaller run with 204 tasks. Each task in the graph is labeled with its program order.

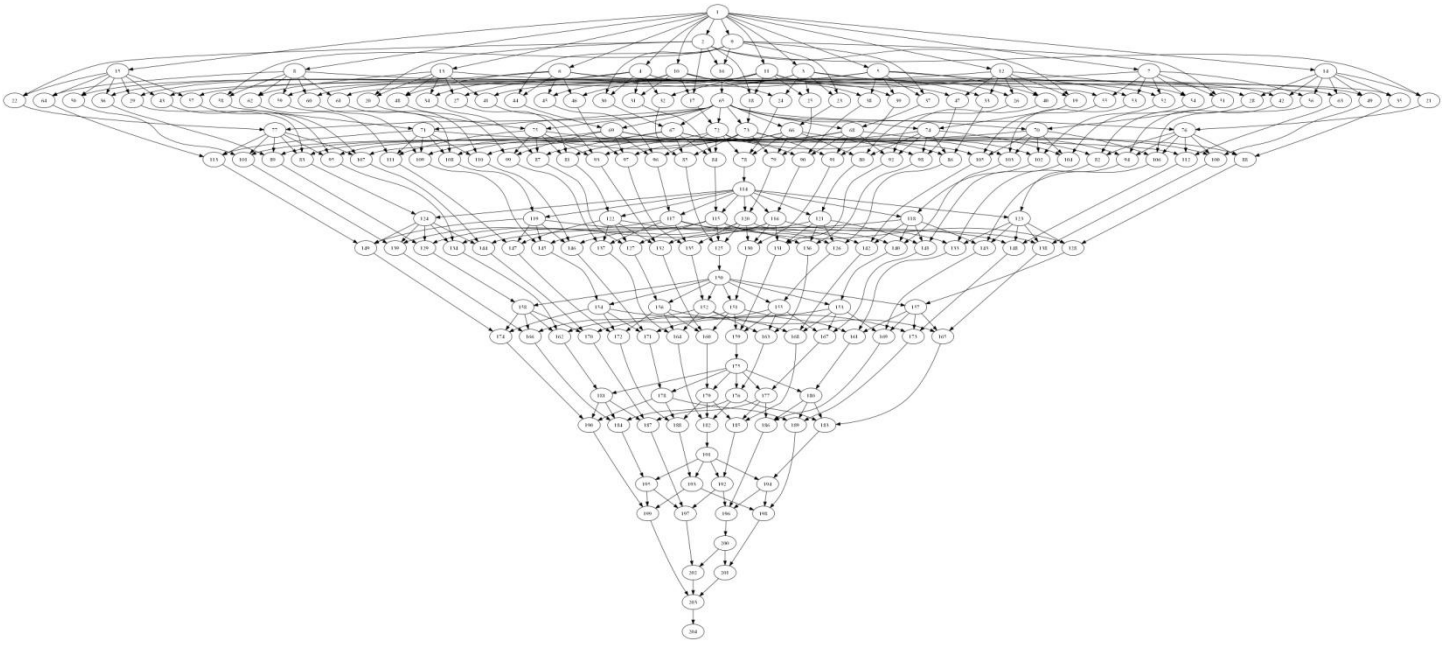

Figure 3.2—1 Sample Dependency graph for the LU application(N256x256,b16x16).

The first parameter we examine is the effect of the task window. In the graph below the x-axis represents the size of the task window ranging from 32 to 16384, while the y axis represents a numerical value. We plot the total number of dependencies labeled “Dependencies”, the number of “Blocking Dependencies” which stand for the number of Dependencies that cause a task to block and the “writebacks” which is the number of writebacks ADAM performed. “Dependencies” as a numerical value include “Blocking Dependencies”.

ADAM performs dependence analysis among the tasks within the task window therefore as the size of the task window increases so does the number of dependencies. Beyond the size of 1024 the number of dependencies remain constant because ADAM has detected all the dependencies in the current run. On the other hand, the writebacks decrease as the size of the task window increases. Beyond the task window size of 1024 because ADAM has identified all the dependencies, ADAM has also identified all the intermediate values, thus avoiding most of the writebacks

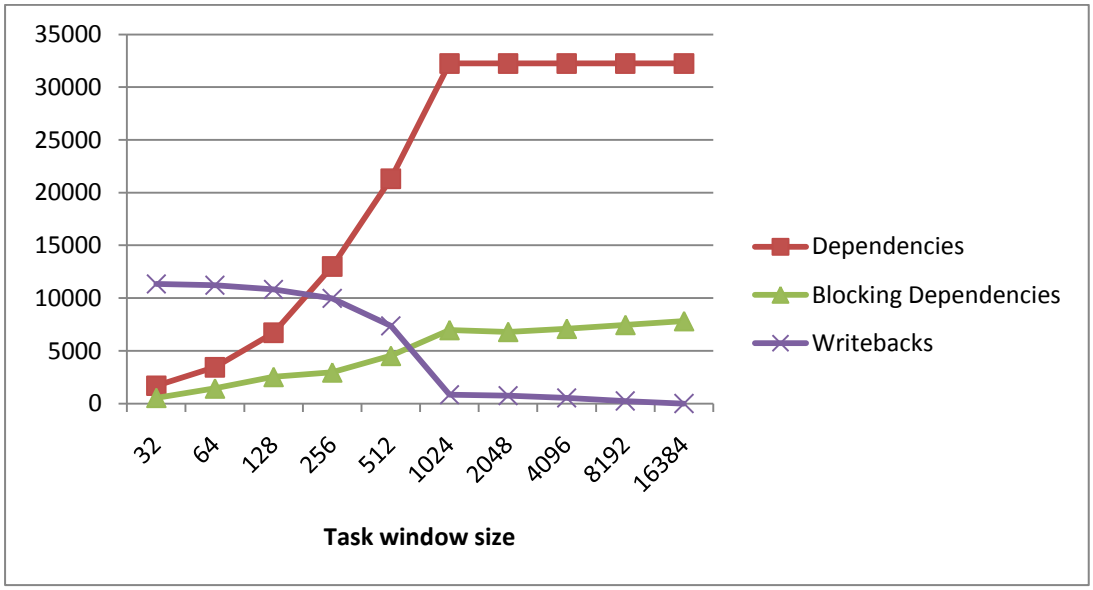

Figure 3.2—2 Task window. The effect of the size of the task window on the number of Dependencies, Blocking Dependencies and Writebacks.

Fewer dependencies translate to less analysis overhead as demonstrated in **Figure 3.2—3**, since the number of dependencies affects the instantiate overhead. While reducing the Task window reduces the number of dependencies and therefore the overhead of the instantiate part, it also increases the overhead due to writebacks. It is up to the programmer to select the appropriate task window that trades-off between these overheads.

The task window size should be high enough so as to enable ADAM to track all dependencies. This high-mark is presented when the number of dependencies becomes stable. In **Figure 3.2—2** this high-mark is 1024. At this high-mark ADAM has tracked all the dependencies of the application and all the intermediate values. Therefore setting the task window at a higher value would not benefit the parallelism or the writeback overhead, it would simply enlarge the runtime's Memory Image.

In certain cases though where the introduced overhead from ADAM is limiting scalability, and the dominant part of the overhead is the instantiate part, it is best to reduce

the task window. Reducing the task window reduces the amount of dependencies detected by ADAM and therefore the instantiate overhead as seen in **Figure 3.2—3**, but it also increases the writeback overhead. In applications that iterate over a working data-set the task-window should be large enough to contain the tasks of one iteration plus one task.

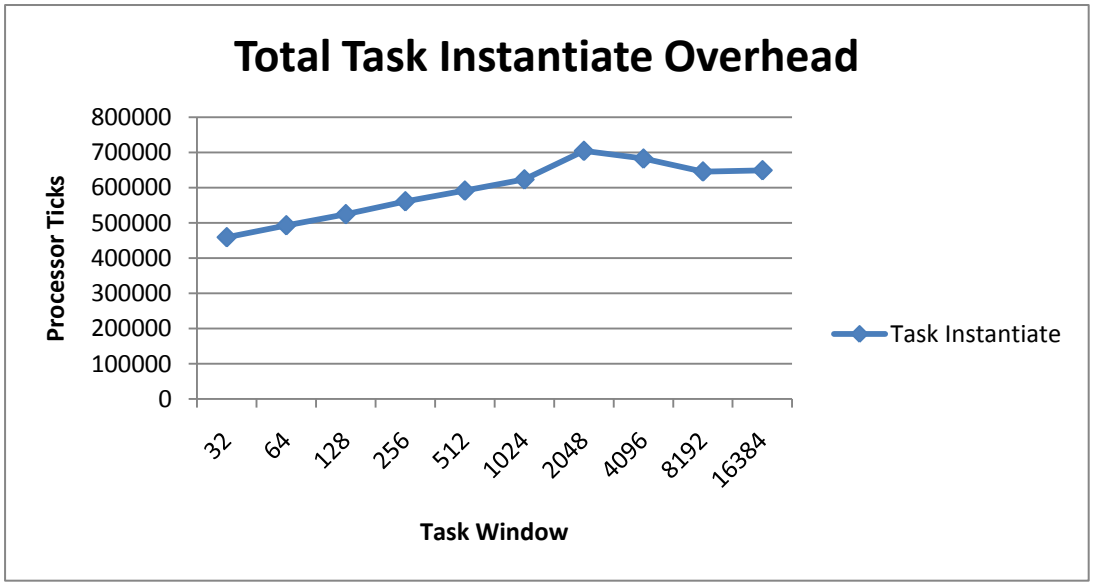

Figure 3.2—3 Task Instantiate vs. Task Window. The Effect of the Task Window size on the task instantiate overhead.

ADAM has the ability to delegate tasks to the Master for execution. In **Figure 3.2—4** we compare the performance of ADAM with the utilization of the Master and without, with the left bar representing ADAM without PPE tasks and the right bar with PPE tasks. The PPE executes tasks whenever ADAM detects a stall. As we can see in the graph the PPE time essentially replaces the stall time and thus the improvement, from PPE tasks, depends on the amount of stall time in the application. **Table 3.2—I** shows the amount of tasks executed by the Master.

PPE tasks are always beneficial for the application. When using PPE tasks we essentially add one more worker to the runtime. Assuming that the application exhibits enough parallelism to utilize an additional worker, the benefit of the PPE tasks depends on the amount of stall time in the application. As we can see in **Figure 3.2—4** in runs with 1 SPE where stall time dominates, the use of PPE tasks yields a speedup of 2,19x. The reason for the super-linear speedup is that PPE-tasks do not require any communication overhead. Because ADAM uses PPE tasks only in cases where it detects stalls, PPE tasks rarely have a negative effect on the application. In fact the only case where an application may not benefit from PPE tasks is if the tasks are large enough to cause lags between scheduler invocations.

Finally because the PPE and the SPEs are of different architectures a task is likely to perform better on one architecture than the other. This can affect the gain from PPE tasks, both in a negative and in a positive manner.

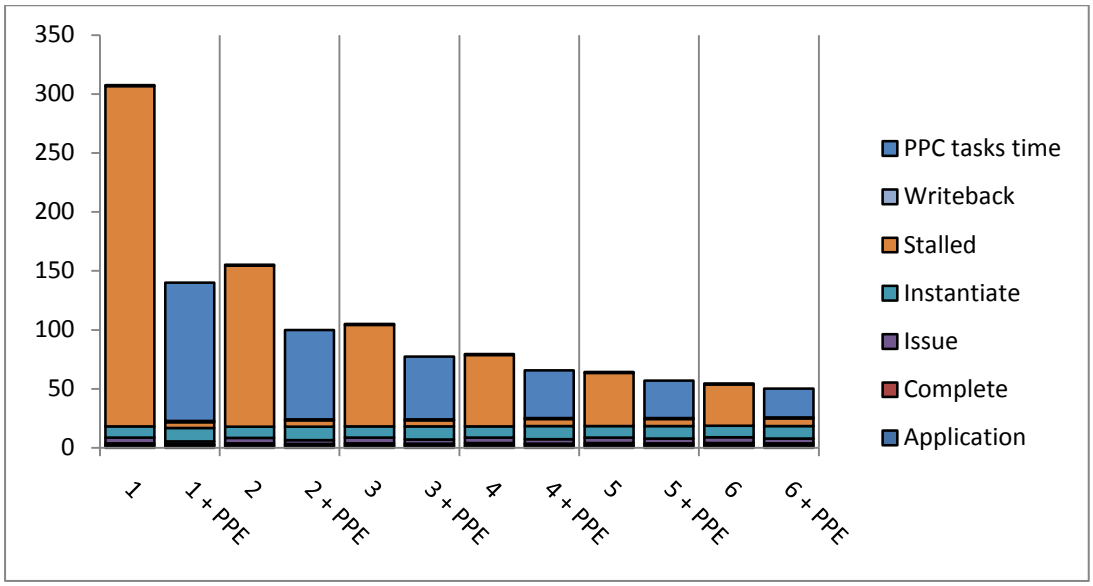

Figure 3.2—4 PPE tasks(LU N512x512,b16x16) . The impact in scalability of the use of PPE tasks.

Runs	Number of PPE tasks	Number of SPE tasks
1SPE +PPE	5345	6095
2SPE +PPE	3417	8023
3SPE +PPE	2397	9043
4SPE +PPE	1779	9661
5SPE +PPE	1383	10057
6SPE +PPE	1050	10390

Table 3.2—I The distribution of Tasks among the PPE and the SPEs

Another parameter that affects the behavior of ADAM is the analysis blocks size. Smaller block sizes increase the granularity of the analysis. In this application the default granularity is 2048 bytes and exports all the available parallelism. We present results with 1024 and 512-byte block size and compare with the default 2048. The graph in **Figure 3.2—5** displays the affect of the block size(x-axis) to the “Dependencies” and the “Blocking Dependencies”

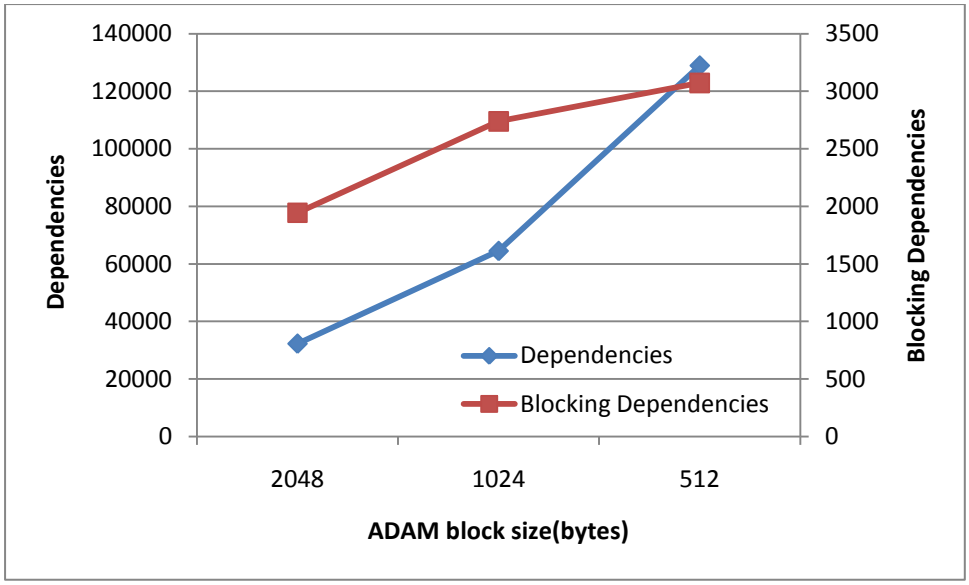

Figure 3.2—5 ADAM block size. The impact of the block size of ADAM in the number of Dependencies and the number of Blocking Dependencies.

The Analysis Block size is a very important parameter and should be chosen with caution. ADAM's minimum supported block size is 16-bytes. Smaller block size usually yields better analysis and ADAM exports more parallelism, however because ADAM is required to process more blocks the overhead is increased. On the other hand large block sizes reduce overhead but may introduce false dependencies, thus limiting parallelism. The block size should match the average argument size of the application, unless the application's scalability is limited by synchronization (stall and wait time). In that case the block size should be reduced allowing ADAM to extract more parallelism if possible. In any case the analysis block size should never be less than the smallest write argument in the application. Below that limit only overhead is introduced.

In the LU application since ADAM performs per block analysis reducing the analysis block in half (1024), doubles the blocks we analyze and therefore duplicates the instantiate and complete overhead.

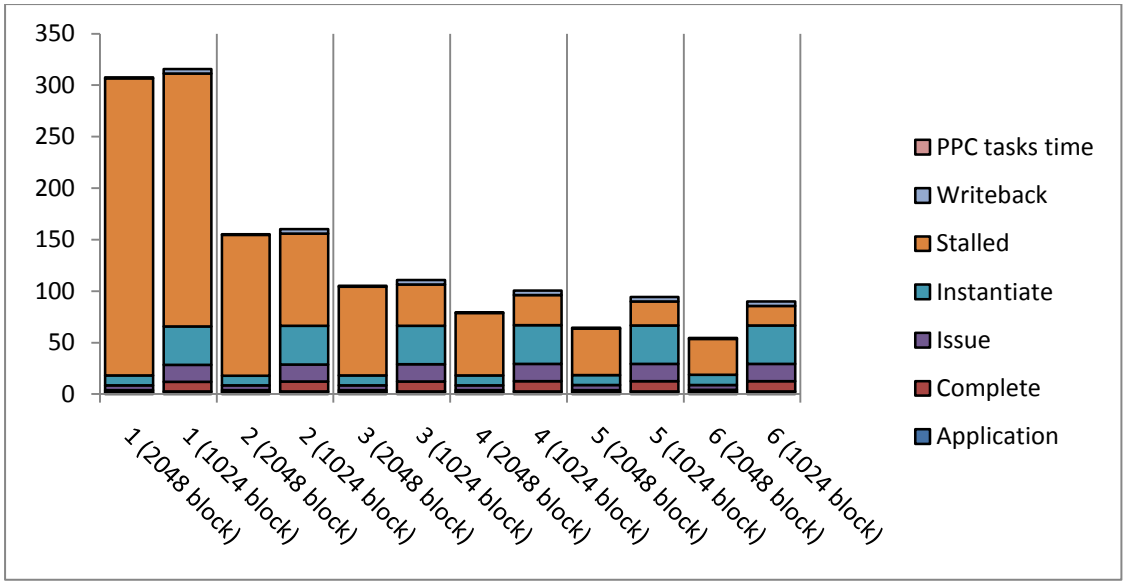

Figure 3.2—6 The impact of ADAM's Block size in scalability (LU N512x512b16x16). Comparison of the PPE breakdown for ADAM block sizes 2048 and 1024 with 1,2,3,4,5,6 SPEs.

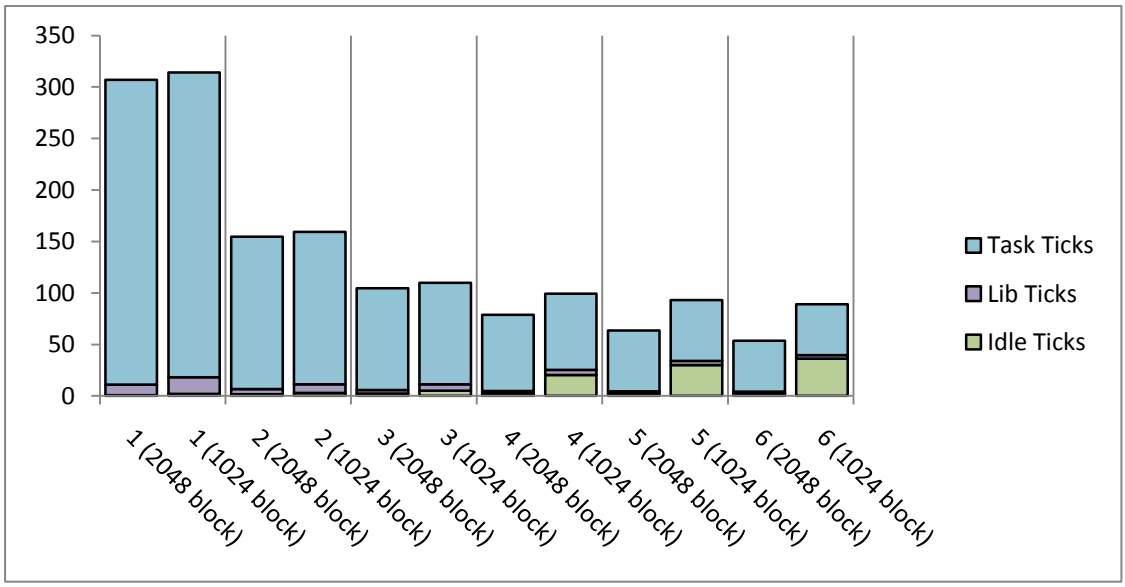

Figure 3.2—7 The impact of ADAM's Block size in scalability (LU N512x512b16x16). Comparison of the SPE breakdown for ADAM block sizes 2048 and 1024 with 1,2,3,4,5,6 SPEs.

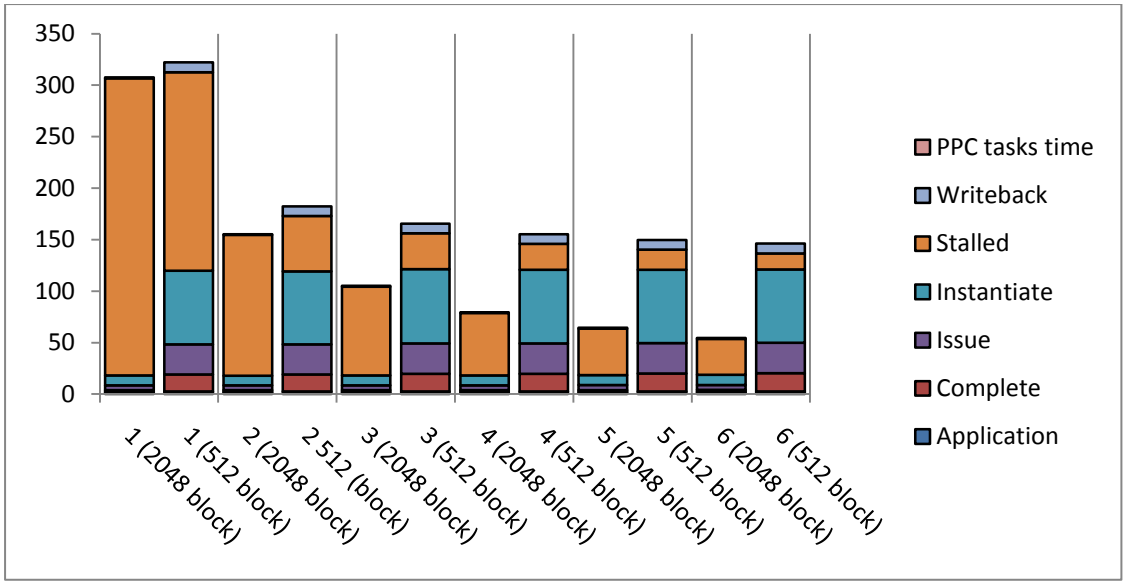

Figure 3.2—8 The impact of ADAM's Block size in scalability (LU N512x512b16x16). Comparison of the PPE breakdown for ADAM block sizes 2048 and 512 with 1,2,3,4,5,6 SPEs.

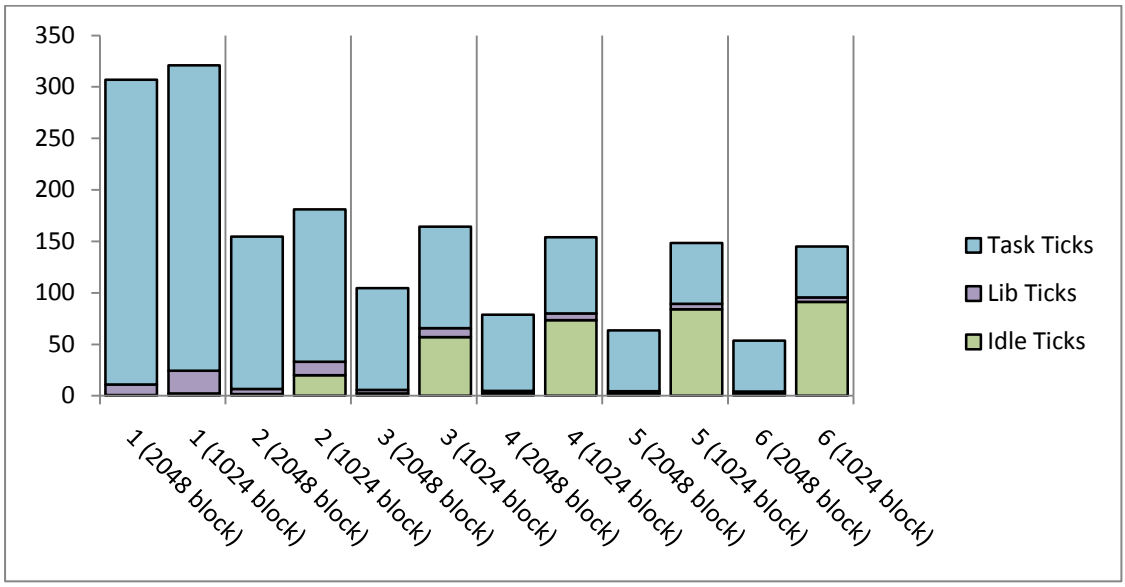

Figure 3.2—9 The impact of ADAM's Block size in scalability (LU N512x512b16x16). Comparison of the SPE breakdown for ADAM block sizes 2048 and 512 with 1,2,3,4,5,6 SPEs.

3.3 Evaluating against CELLSS [2]

We Compare ADAM with CELLSS [**2**] using the benchmarks that originate from the CELLSS [**2**] runtime. These include Cholesky, Matmul and Jacobi and for each of these we use the same configurations as in the first section of the evaluation. Because the CELLSS runtime does not produce execution breakdowns we only compare total execution times excluding initialization times.

3.3.1 Cholesky

In the Cholesky benchmark, ADAM outperforms CELLSS [**2**] and yields speedups of 2,77 times faster for the 13x13 dataset and 6 SPEs, and 1,83 times faster for the 20x20 than the corresponding CELLSS runs. ADAM performs better in Cholesky due to lower analysis overheads. Because this is a computation intensive benchmark the significance of the analysis overheads is related to the data-set size. As the data-set grows the overheads become less apparent on the overall execution time and therefore the difference between the two runtimes decreases. As the data-set shrinks however, ADAM's speedup over CELLSS [**2**] increases.

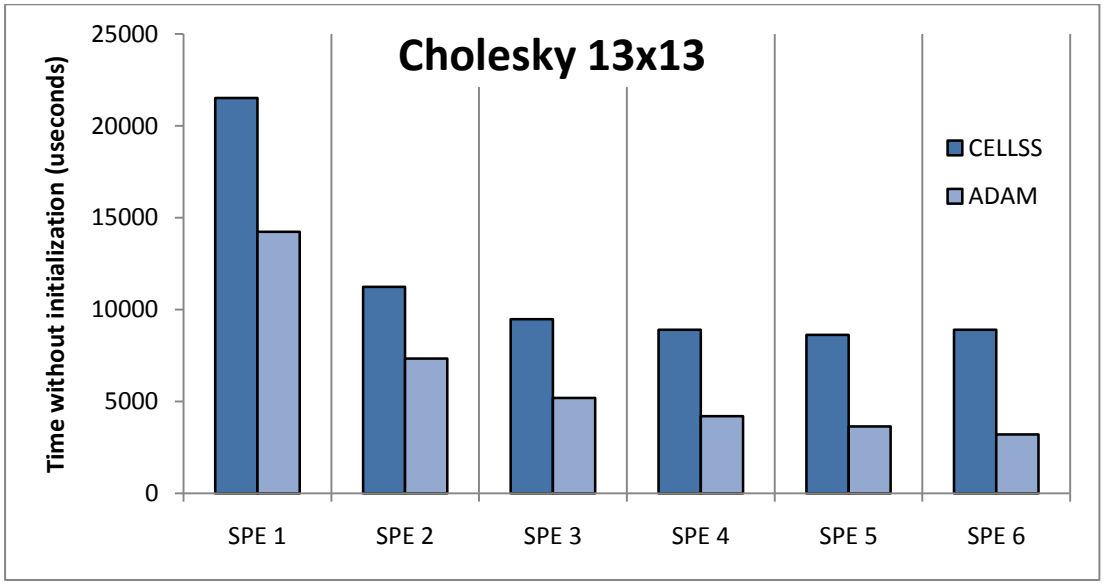

Figure 3.3.1—1 Cholesky 13x13 ADAM vs. CELLSS. Cholesky for runs with 1,2,3,4,5,6 SPEs and for a 13x13 matrix of 64x64 blocks(Cholesky blocks) of single precision numbers.

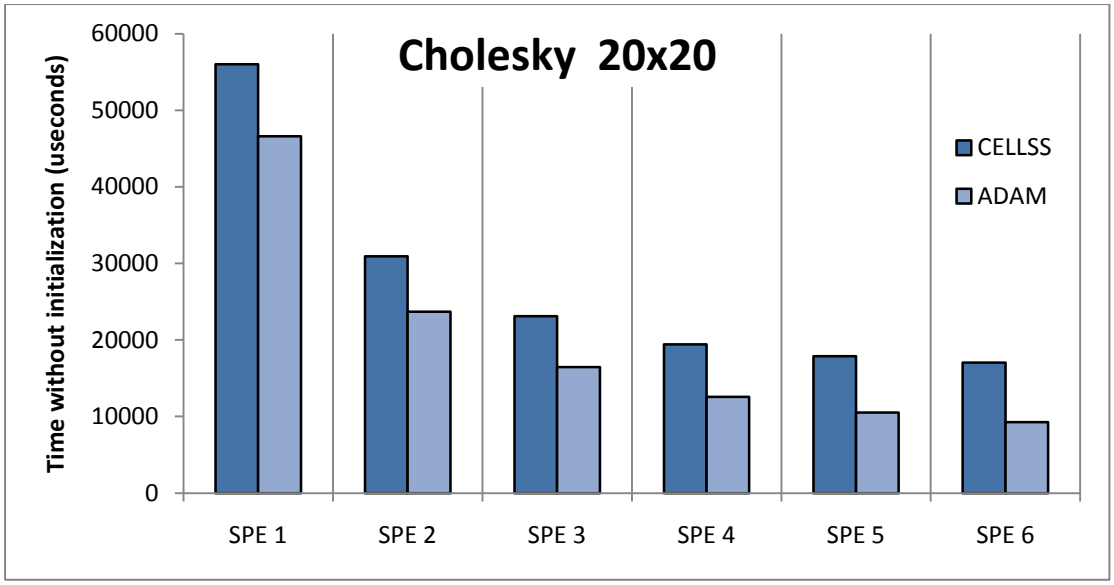

Figure 3.3.1—2 Cholesky 20x20 ADAM vs. CELLSS. Cholesky for runs with 1,2,3,4,5,6 SPEs and for a 20x20 matrix of 64x64 blocks(Cholesky blocks) of single precision numbers.

3.3.2 Matmul

Matmul is also a computation intensive benchmark. The dependency graph of this application is less complicated than Cholesky, which translates to lower overheads for both runtimes. ADAM however in both **Figure 3.3.2—1** and **Figure 3.3.2—2** maintains a steady advantage.

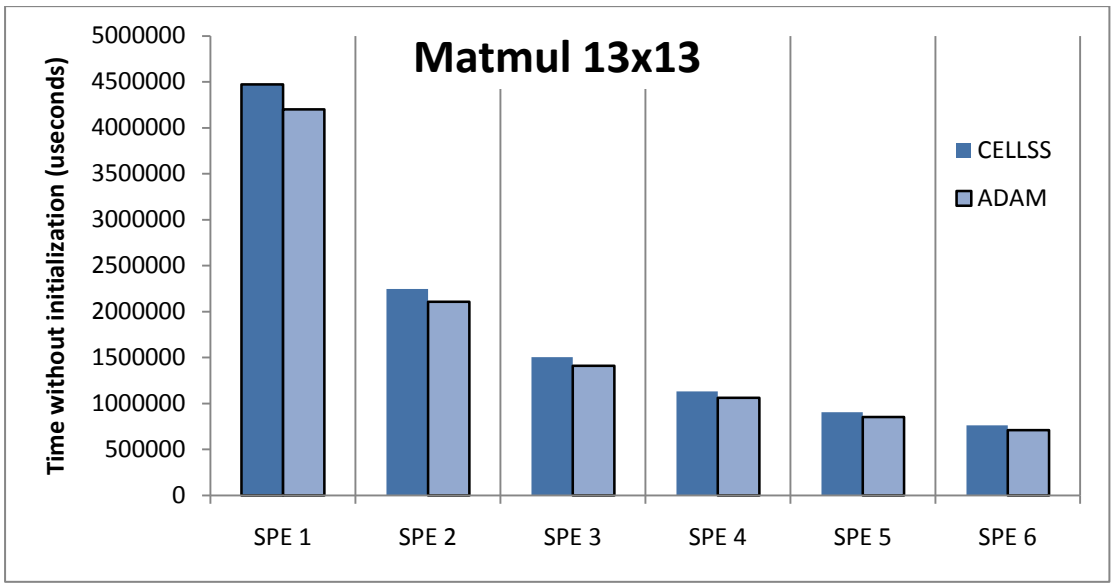

Figure 3.3.2—1 Matmul 13x13 ADAM vs. CELLSS. Matmul for runs with 1,2,3,4,5,6 SPEs and for a 13x13 matrix of 64x64 blocks(Matmul blocks) of single precision numbers.

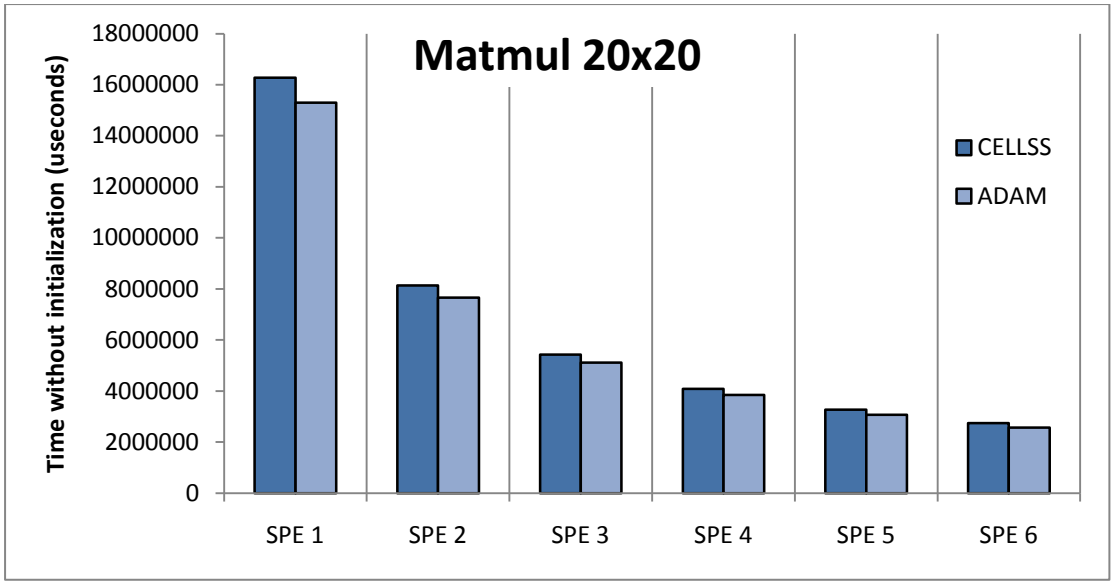

Figure 3.3.2—2 Matmul 20x20 ADAM vs. CELLSS. Matmul for runs with 1,2,3,4,5,6 SPEs and for a 20x20 matrix of 64x64 blocks(Matmul blocks) of single precision numbers.

3.3.3 Jacobi

Jacobi is a communication intensive benchmark that benefits from data renaming. ADAM has lower overheads for dependence analysis and for renaming than CELLSS [**2**]. Furthermore ADAM has an aggressive policy of always renaming. ADAM outperforms the CELLSS [**2**] runtime although ADAM is not able to scale well for the Jacobi application. The speedup of ADAM over the CELLSS [**2**] runtime is 2,27 for 13 iterations with 6 workers and 2,14 for 20 iterations with 6 workers.

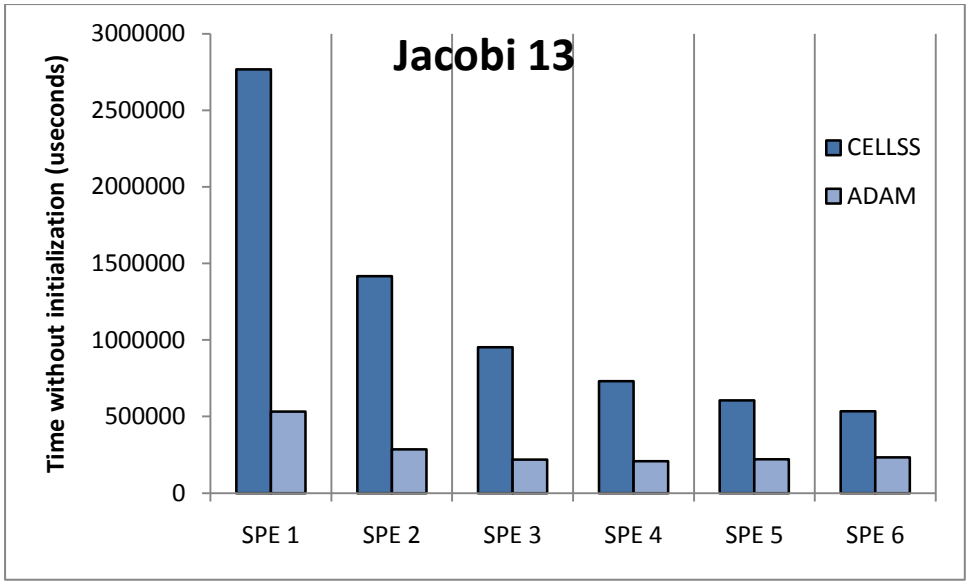

Figure 3.3.3—1 Jacobi 13 ADAM vs. CELLSS. Jacobi for runs with 1,2,3,4,5,6 SPEs and for 13 iterations over a 32x32 matrix of 32x32 blocks(jacobi blocks) of single precision numbers.

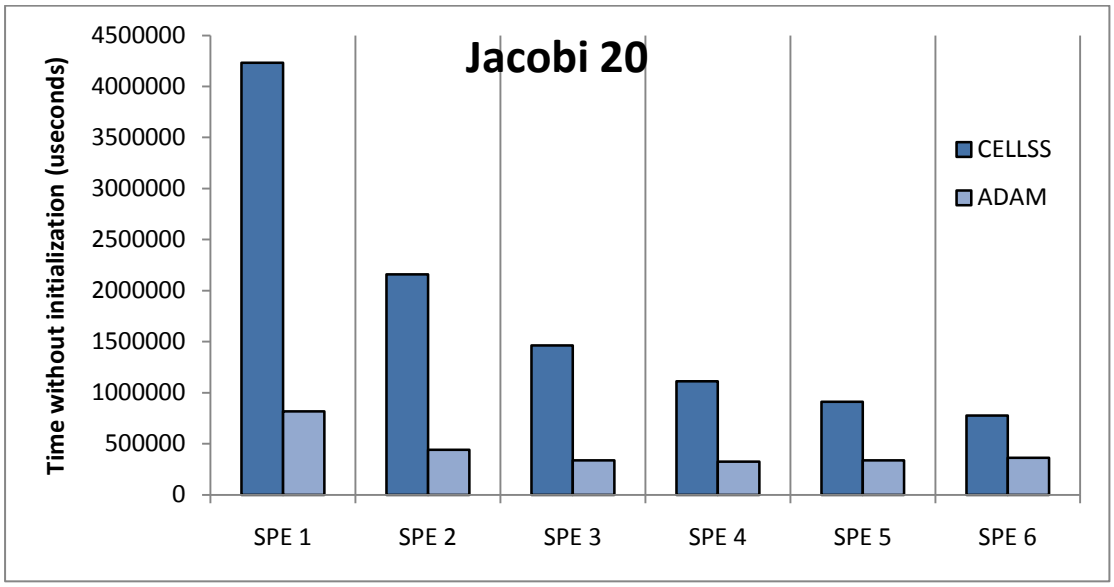

Figure 3.3.3—2 Jacobi 20 ADAM vs. CELLSS. Jacobi for runs with 1,2,3,4,5,6 SPEs and for 20 iterations over a 32x32 matrix of 32x32 blocks(jacobi blocks) of single precision numbers.

3.4 Evaluating against TPC [3]

In this section we compare the performance of ADAM against that of TPC [**3**]. For each application that we evaluate, we present runs that correspond to 6 workers(SPEs) and compare the PPE breakdown(left) to the SPE breakdown(right). The breakdown for the TPC [**3**] corresponds to the following:

TPC PPE Breakdown

- **Issue**
  Issue is the time required for the master to send the task descriptions to the workers. The task description is send via remote stores.
- **Stalled**
  This is the portion of time in which the Master has tasks eligible for execution and polls the worker queues for an available slot.
- **Wait**
  This is the measured time of the Master blocking until workers complete all issued tasks. This time corresponds to barrier synchronization time in TPC. ADAM requires only one barrier at the end of the application.
- **Application**
  This is the time dedicated to the running application excluding the aforementioned overheads and initialization.

TPC SPE Breakdown

- **Task Ticks**
  Task is the portion of time the SPE spends executing task code.
- **Lib Ticks**
  Lib is the portion of time the SPE spends executing library code.
- **Idle Ticks**
  Idle is the time spent by an SPE when it has no pending or executing tasks.

In the cases where the evaluated application's scalability limiting factor is ADAM's overhead, TPC [**3**] performs better than ADAM. In any other case ADAM either performs better or at least matches the performance of the TPC [**3**] runtime.

3.4.1 LU

In the double precision version of LU, ADAM's performance compared to TPC [**3**] is analogous to the LU block size. In runs with 8x8 blocks(**Figure 3.4.1—1,Figure 3.4.1—4**) TPC [**3**] is significantly faster. In theses runs ADAMs introduced overhead(instantiate +

Complete) supersedes task communication and computation time(Lib + task). In the runs with block size 16x16(**Figure 3.4.1—2, Figure 3.4.1—5**) where the task size compensates for the introduced overhead ADAM's performance is slightly better than the TPC [**3**]. ADAM's performance gain against the TPC [**3**] runtime is attributed to the elimination of th synchronization time(wait) and SPE idle time on runs with TPC [**3**]. In runs with larger block sizes (**Figure 3.4.1—3,Figure 3.4.1—6**), for the 512x512 dataset(**Figure 3.4.1—3**) ADAM's performance gain increases because the synchronization(wait) and SPE idle time in the TPC [**3**] runs are increased. Furthermore in these runs we observe that ADAM saturates the workers while converting the uncompensated synchronization time(wait) to stall time. This is an indication that ADAM has extracted more parallelism, and thus could scale to a greater number of workers. For the 4096x4096 dataset **Figure 3.4.1—6** ADAM's performance matches the performance of the TPC [**3**] because the impact of the synchronization time in the TPC [**3**] runs is so minimal that the workers appear saturated. In the single precision runs of LU where the size of the tasks is smaller ADAM is outperformed in runs with 8x8(**Figure 3.4.1—7,Figure 3.4.1—10**) and 16x16(**Figure 3.4.1—8,Figure 3.4.1—11**) LU block sizes. In these runs the ADAM's introduced overheads exceed the corresponding task and communication times(Lib + Task).In the run with 32x32(**Figure 3.4.1—9,Figure 3.4.1—12**) ADAM's behavior is analogous to the corresponding double precision runs (**Figure 3.4.1—3,Figure 3.4.1—6**) and therefore the same observations are valid.

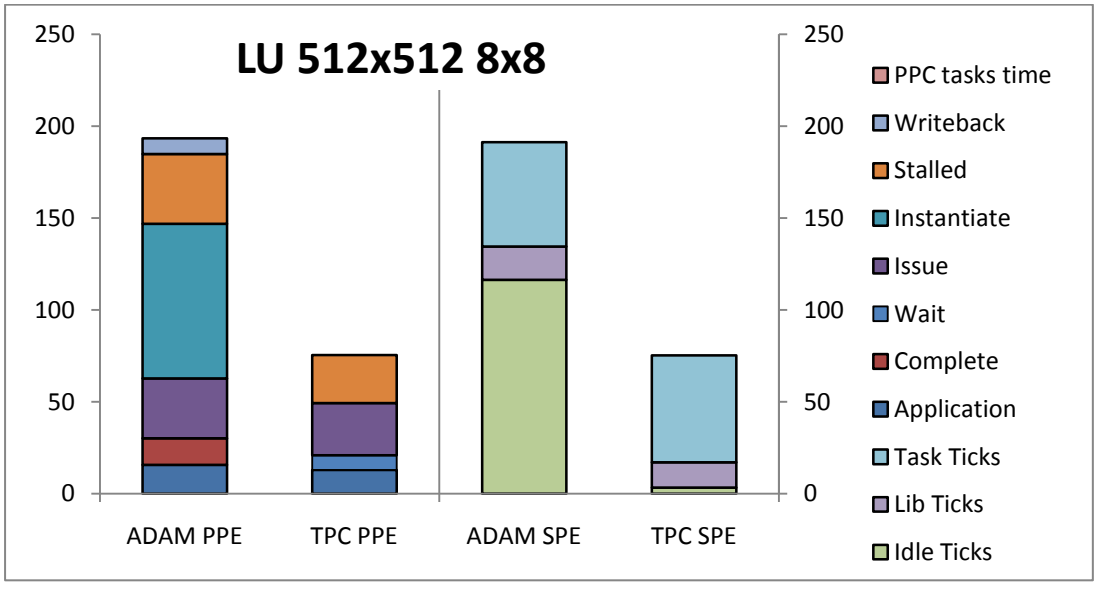

Figure 3.4.1—1 ADAM vs. TPC LU 512x512(8x8). LU execution Breakdown for the PPE and the SPEs for runs with 6 SPEs, for an array size of 512x512 with application block size 8x8.

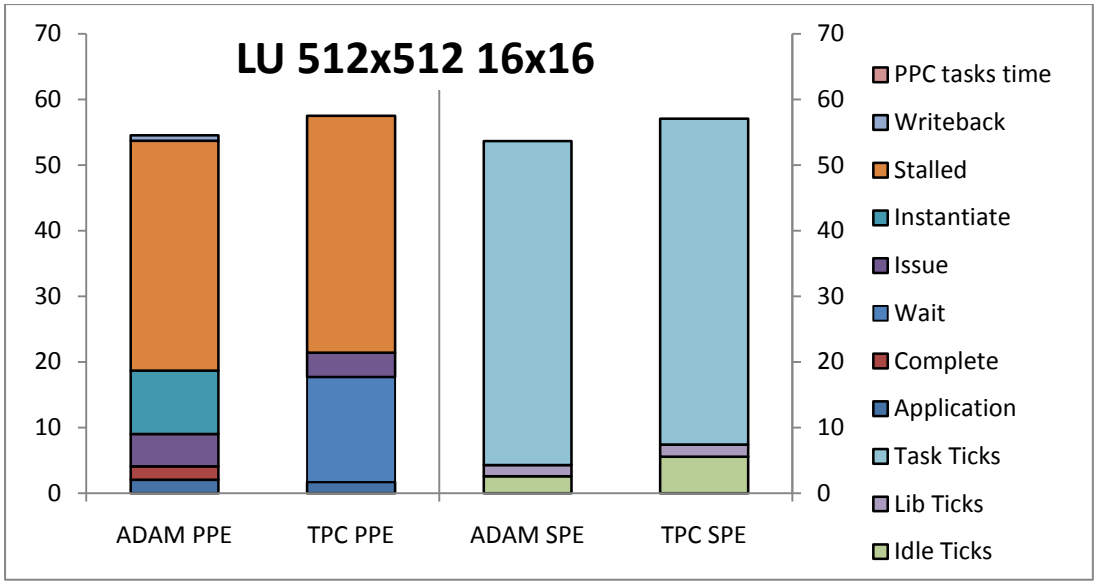

Figure 3.4.1—2 ADAM vs. TPC LU 512x512(16x16). LU execution Breakdown for the PPE and the SPEs for runs with 6 SPEs, for an array size of 512x512 with application block size 16x16.

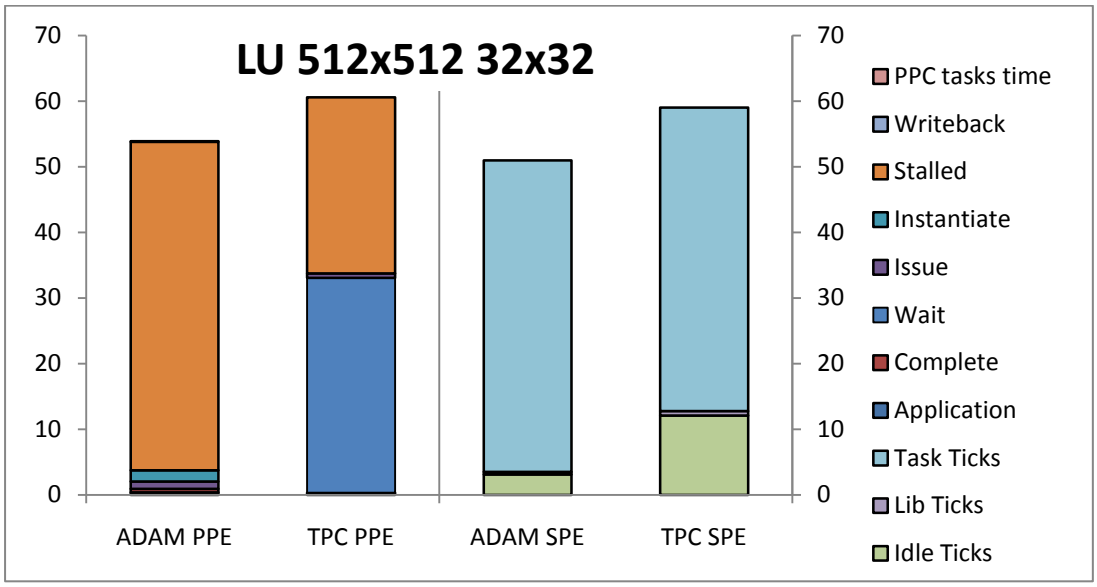

Figure 3.4.1—3 ADAM vs. TPC LU 512x512(32x32). LU execution Breakdown for the PPE and the SPEs for runs with 6 SPEs, for an array size of 512x512 with application block size 32x32.

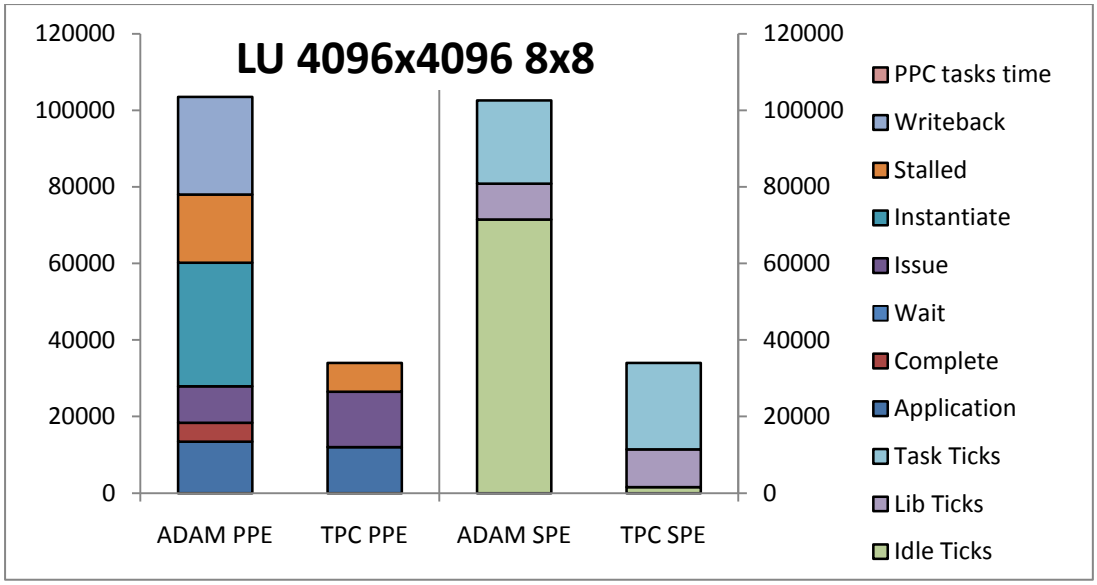

Figure 3.4.1—4 ADAM vs. TPC LU 4096x4096(8x8). LU execution Breakdown for the PPE and the SPEs for runs with 6 SPEs, for an array size of 4096x4096 with application block size 8x8.

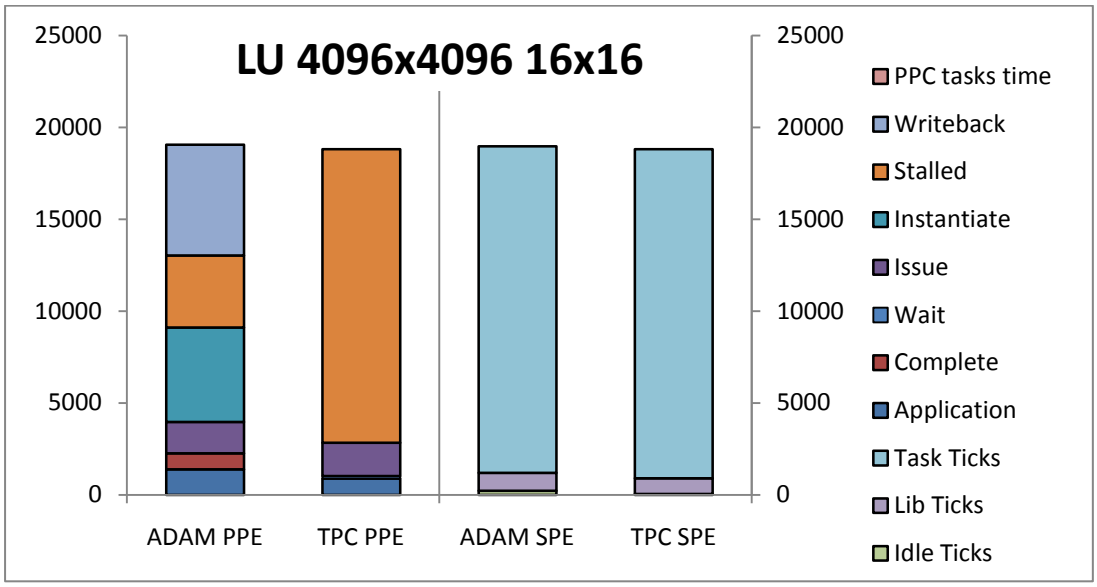

Figure 3.4.1—5 ADAM vs. TPC LU 4096x4096(16x16). LU execution Breakdown for the PPE and the SPEs for runs with 6 SPEs, for an array size of 4096x4096 with application block size 16x16.

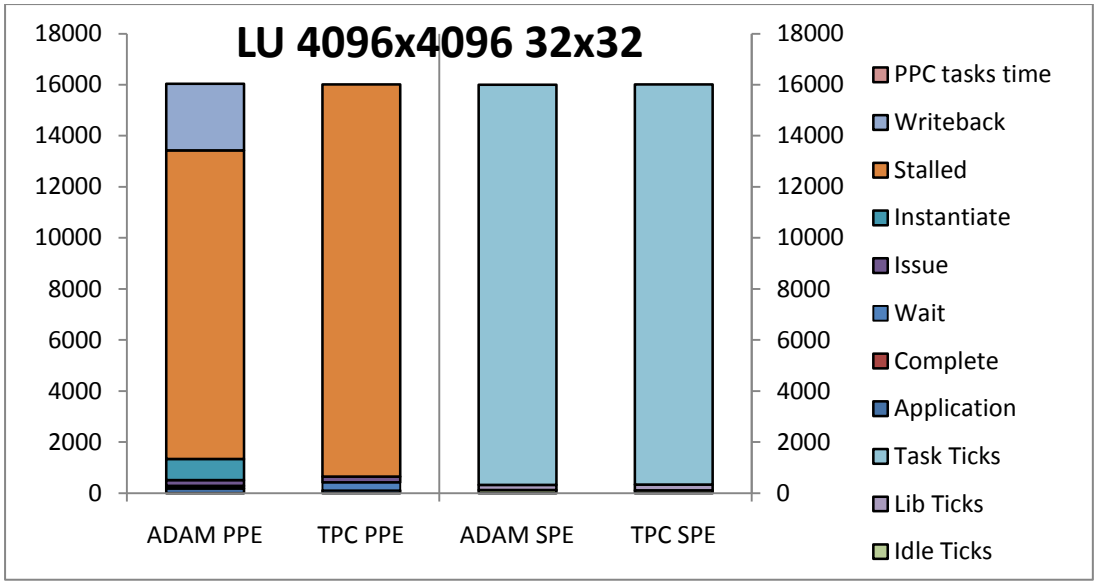

Figure 3.4.1—6 ADAM vs. TPC LU 4096x4096(32x32). LU execution Breakdown for the PPE and the SPEs for runs with 6 SPEs, for an array size of 4096x4096 with application block size 32x32.

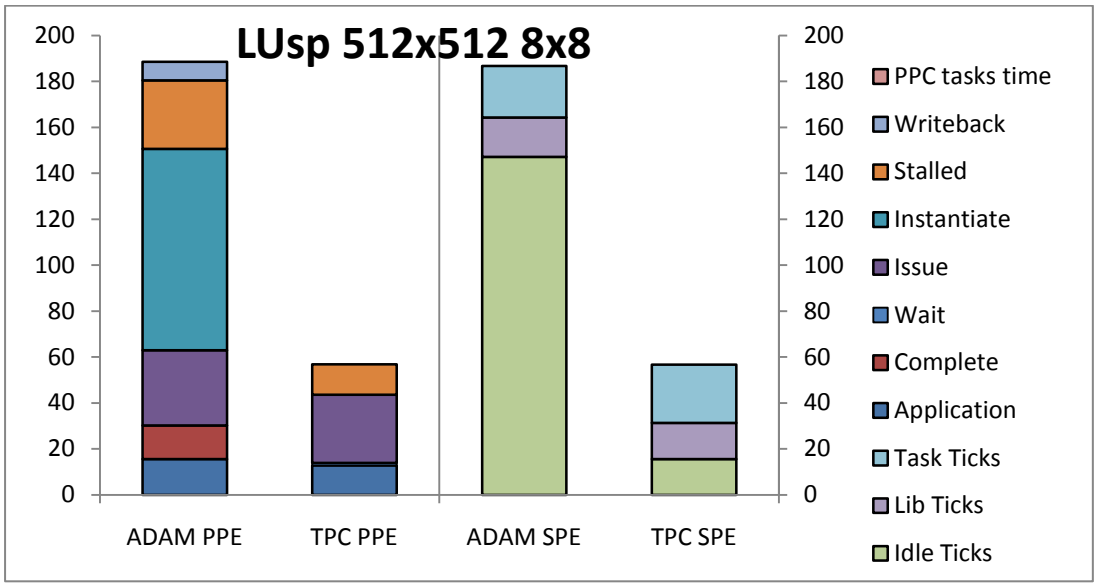

Figure 3.4.1—7 ADAM vs. TPC LUsp 512x512(8x8). LU with single precision numbers execution Breakdown for the PPE and the SPEs for runs with 6 SPEs, for an array size of 512x512 with application block size 8x8.

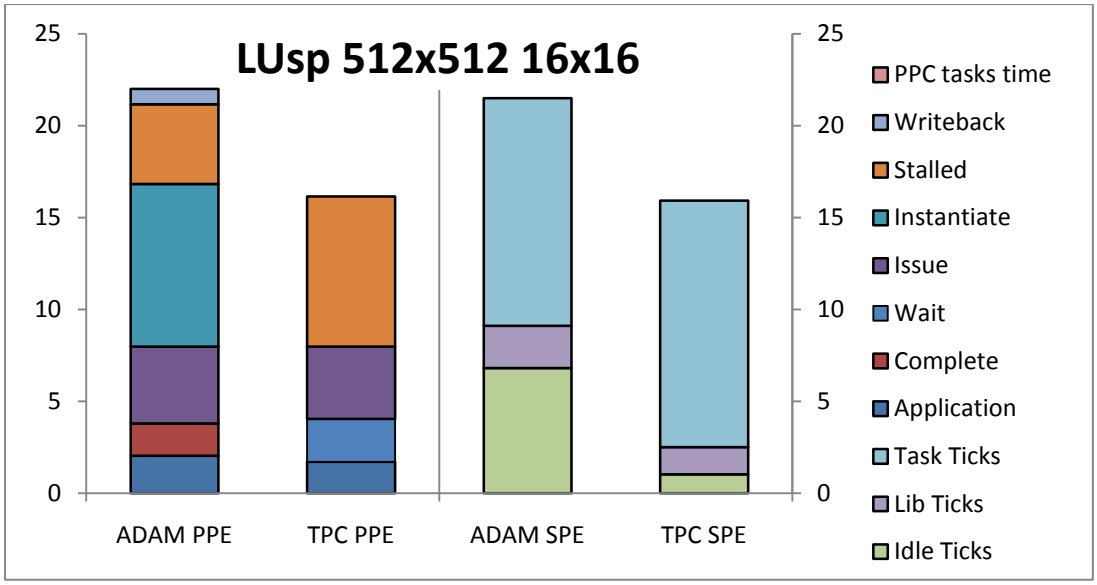

Figure 3.4.1—8 ADAM vs. TPC LUsp 512x512(16x16). LU with single precision numbers execution Breakdown for the PPE and the SPEs for runs with 6 SPEs, for an array size of 512x512 with application block size 16x16.

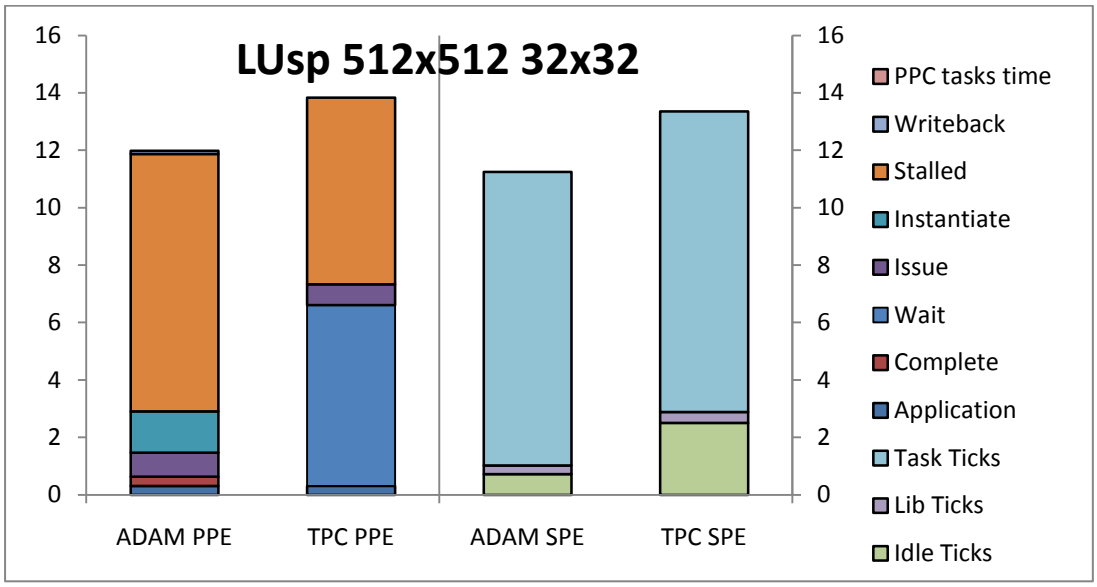

Figure 3.4.1—9 ADAM vs. TPC LUsp 512x512(32x32). LU with single precision numbers execution Breakdown for the PPE and the SPEs for runs with 6 SPEs, for an array size of 512x512 with application block size 32x32.

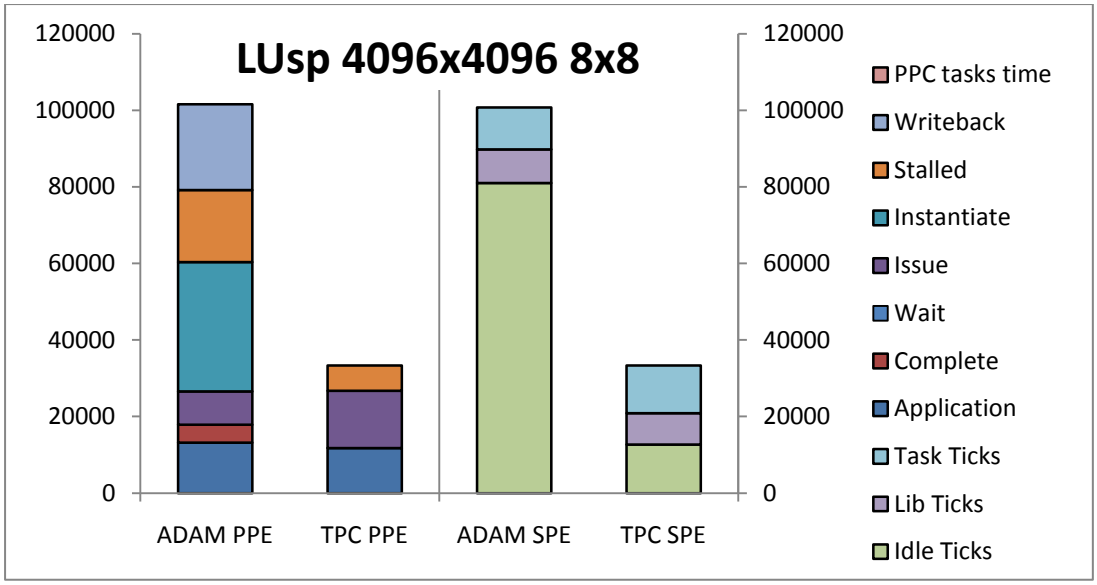

Figure 3.4.1—10 ADAM vs. TPC LUsp 4096x4096(8x8). LU with single precision numbers execution Breakdown for the PPE and the SPEs for runs with 6 SPEs, for an array size of 4096x4096 with application block size 8x8.

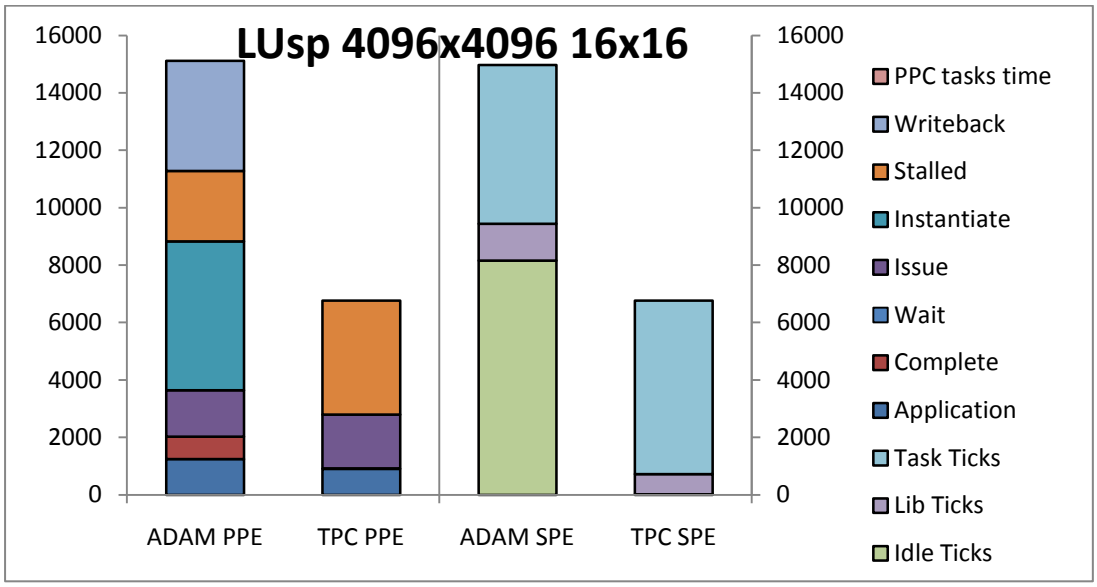

Figure 3.4.1—11 ADAM vs. TPC LUsp 4096x4096(16x16). LU with single precision numbers execution Breakdown for the PPE and the SPEs for runs with 6 SPEs, for an array size of 4096x4096 with application block size 16x16.

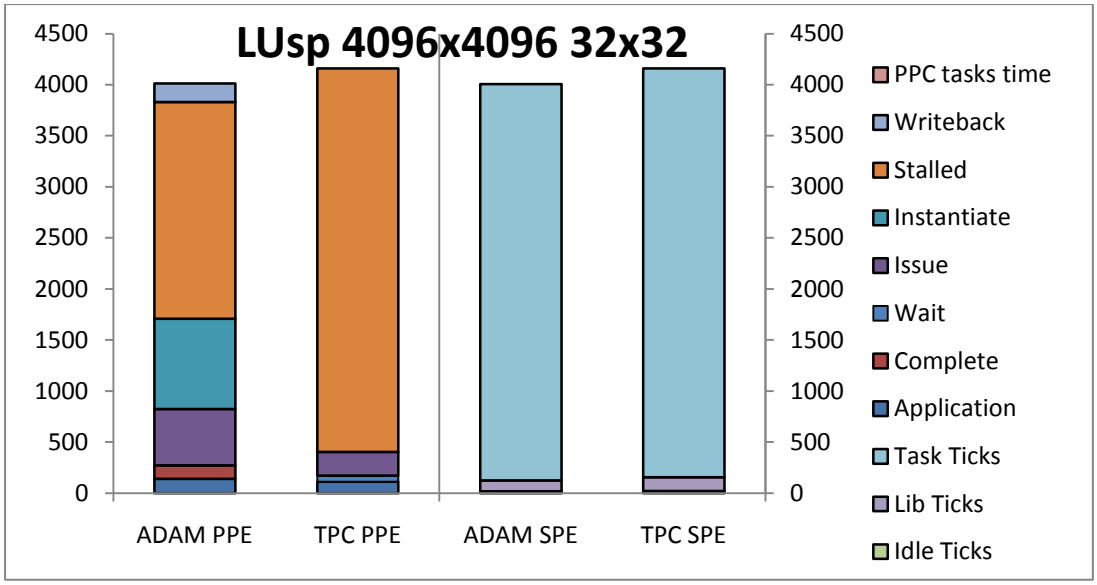

Figure 3.4.1—12 ADAM vs. TPC LUsp 4096x4096(32x32). LU with single precision numbers execution Breakdown for the PPE and the SPEs for runs with 6 SPEs, for an array size of 4096x4096 with application block size 32x32.

3.4.2 FFT

For the FFT benchmark TPC [**3**] achieves better performance than ADAM in both the double precision runs(**Figure 3.4.2—1,Figure 3.4.2—2**) and the single precision runs(**Figure 3.4.2—3,Figure 3.4.2—4**). In the double precision FFT runs with 64k and 4M (**Figure 3.4.2—1, Figure 3.4.2—2)** although the introduced aggregated overhead is less than the aggregated task communication and computation time, ADAM introduces Idle time to the SPEs. This is because the tasks used for the transposition phase have strided arguments. ADAM analyzes each element as a separate argument, which explains why the issue time is higher compared to the TPC [**3**]. For the tasks with strided arguments the introduced overhead supersedes task times thus resulting in SPE Idle time. In the single precision runs (**Figure 3.4.2—3,Figure 3.4.2—4**) ADAM's introduced overheads dominate the execution times.

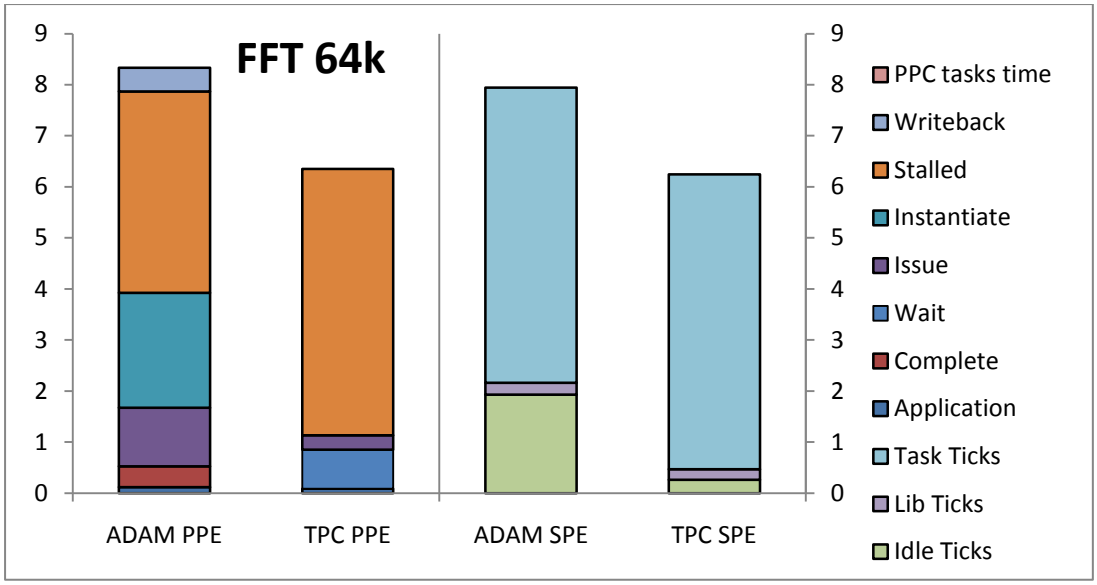

Figure 3.4.2—1 ADAM vs. TPC FFT 64k. FFT, execution Breakdown for the PPE and the SPEs for runs with 6 SPEs, for 16384 Complex Double elements.

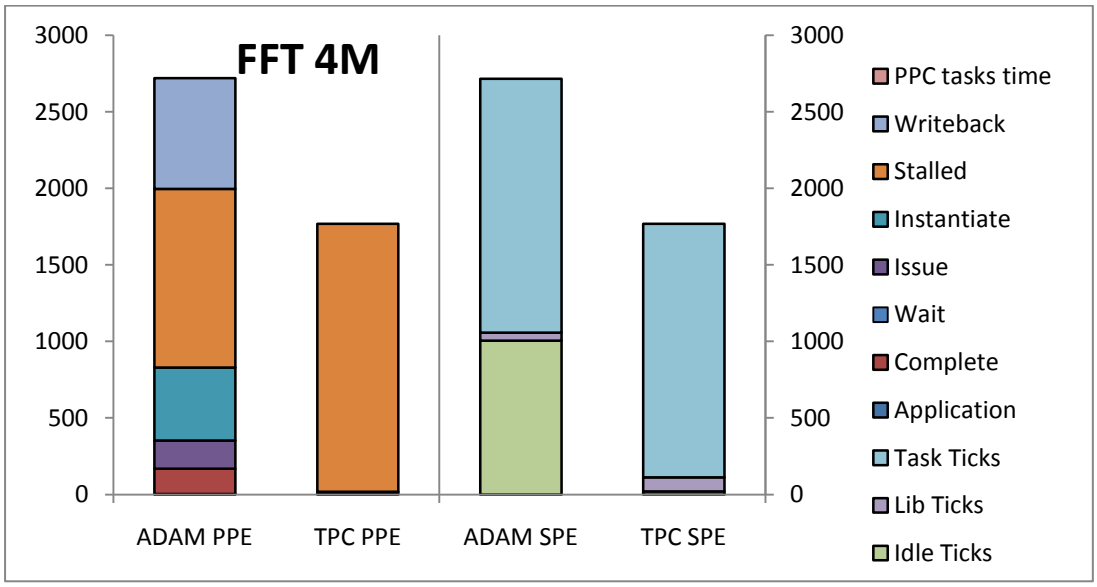

Figure 3.4.2—2 ADAM vs. TPC FFT 4M. FFT execution Breakdown for the PPE and the SPEs for runs with 6 SPEs, for 4194304 Complex Double elements.

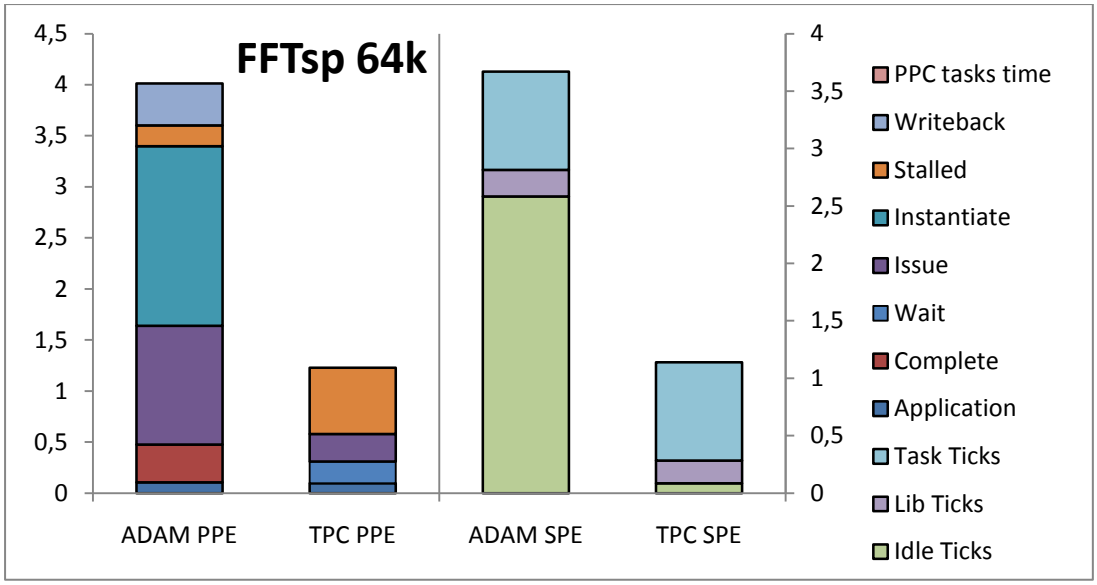

Figure 3.4.2—3 ADAM vs. TPC FFTsp 64k. FFT with single precision, execution Breakdown for the PPE and the SPEs for runs with 6 SPEs, for 16384 Complex Float elements.

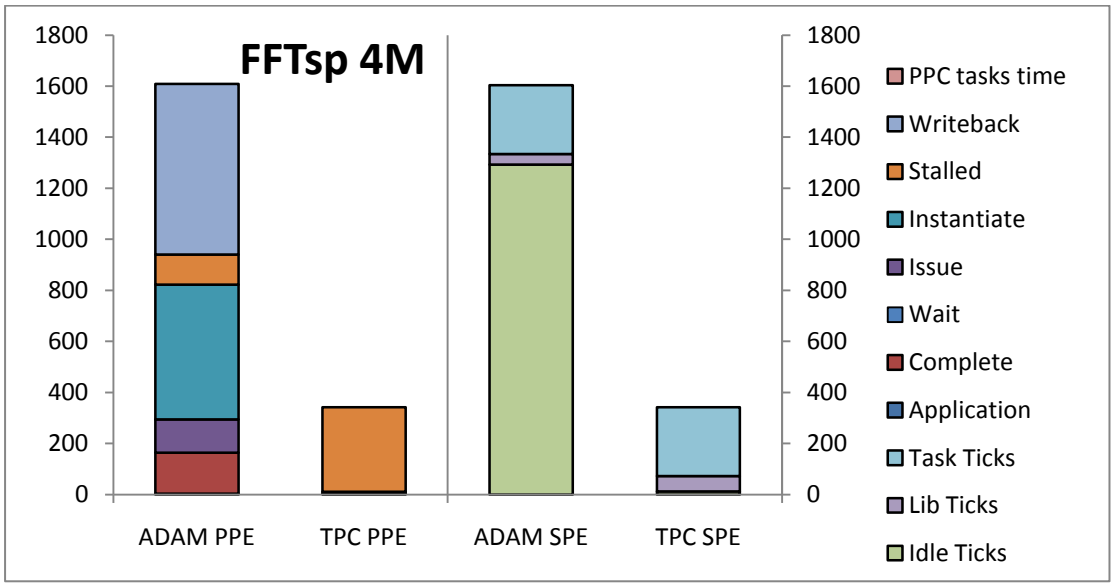

Figure 3.4.2—4 ADAM vs. TPC FFTsp 4M. FFT with single precision, execution Breakdown for the PPE and the SPEs for runs with 6 SPEs, for 4194304Complex Float elements.

3.4.3 Sequoia [4] Kernels

For the sequioa [**4**] micro-benchmarks ADAM performs significantly lower than the TPC runitme [**3**]. The task size in these benchmarks is extremely low, while the number of tasks excessively large, which causes the overheads to dominate. The low execution time compared to the number of tasks causes small differences in same label overheads to scale-up.

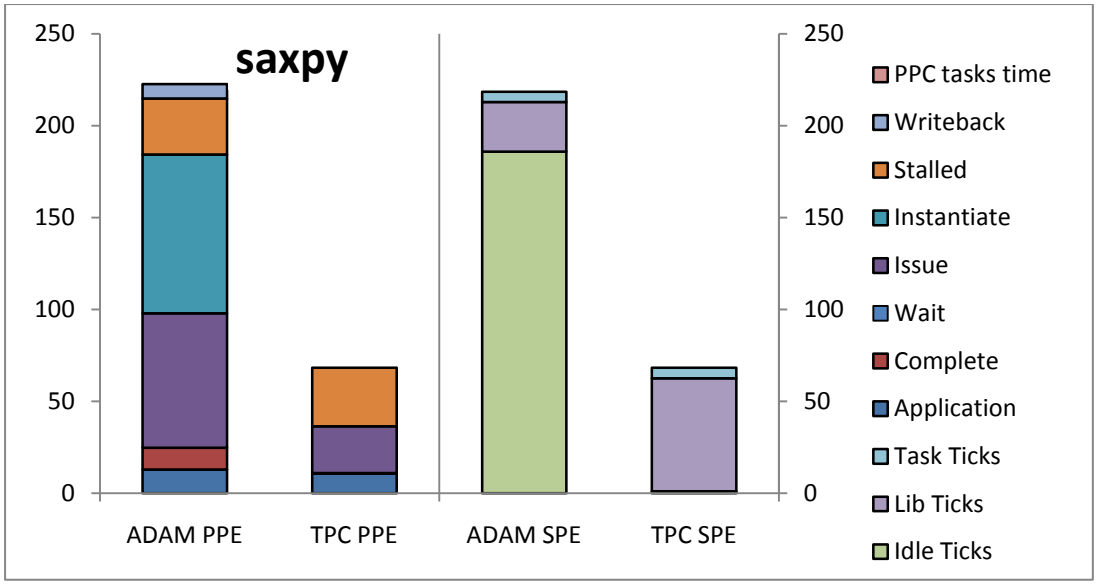

Figure 3.4.3—1 ADAM vs. TPC SAXPY. Saxpy execution Breakdown for the PPE and the SPEs for runs with 6 SPEs.

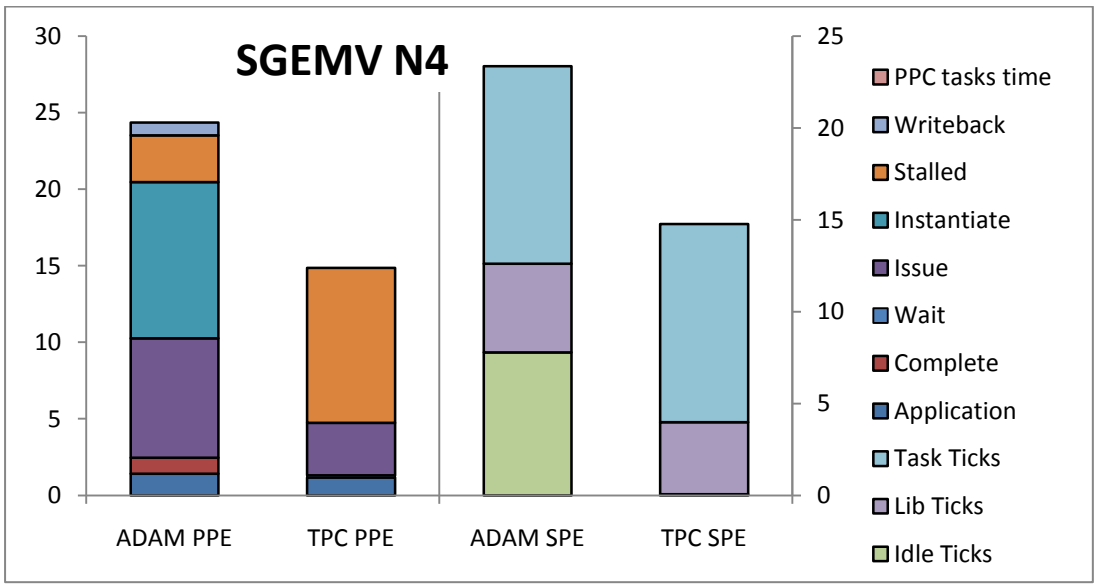

Figure 3.4.3—2 ADAM vs. TPC SGEMV N4. Saxpy execution Breakdown for the PPE and the SPEs with N=4 for runs with 6 SPEs.

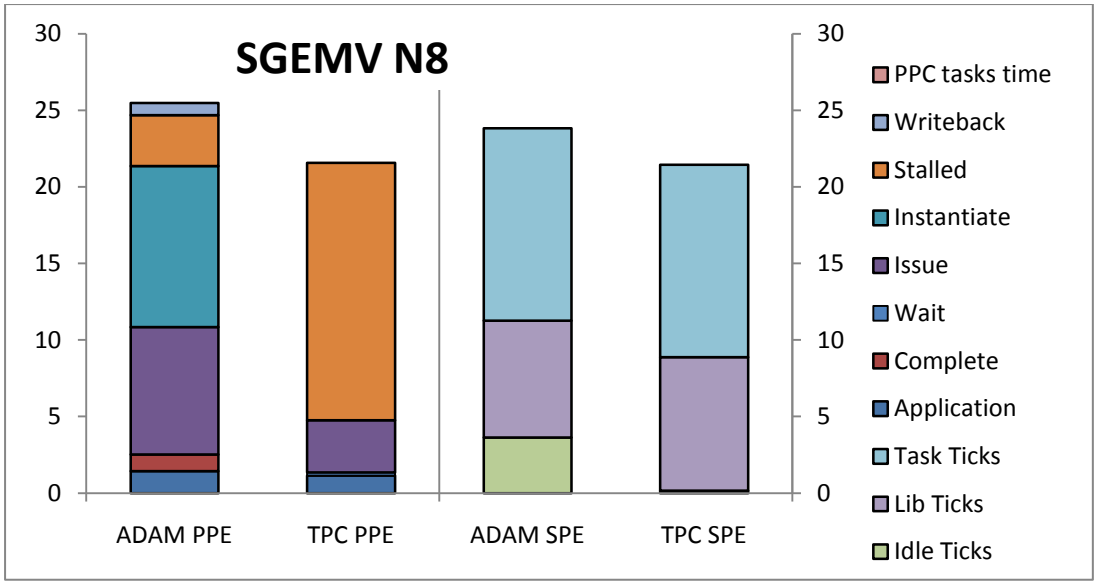

Figure 3.4.3—3 ADAM vs. TPC SGEMV N8. Saxpy execution Breakdown for the PPE and the SPEs with N=8 for runs with 6 SPEs.

3.4.4 Cholesky

In the Cholesky Benchmark ADAM performs significantly better than the TPC [**3**] runtime for both the 13x13(**Figure 3.4.4—1**) and the 20x20(**Figure 3.4.4—2**) dataset. The introduced overheads by ADAM do not affect the overall performance. Due to the dependencies among tasks Cholesky requires a lot of synchronization in the TPC [**3**] runs which in fact dominates the TPC [**3**] execution times, and creates SPE idle time. ADAM through dependence analysis eliminates synchronization time and thus achieves better performance.

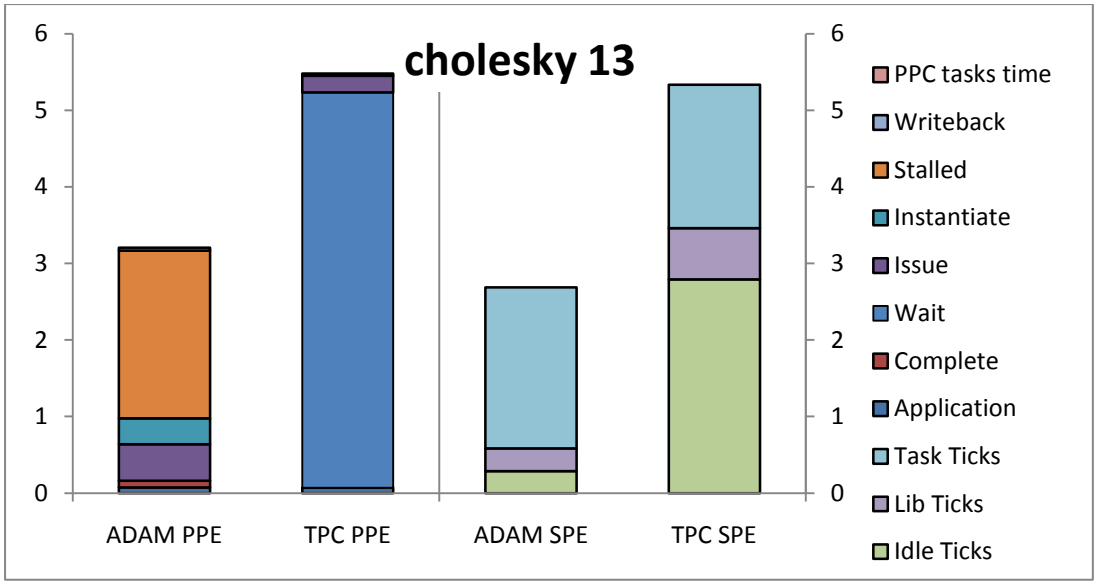

Figure 3.4.4—1 ADAM vs. TPC Cholesky 13x13. Cholesky execution Breakdown for the PPE and the SPEs for runs with 6 SPEs and for a 13x13 matrix of 64x64 blocks(Cholesky blocks) of single precision numbers.

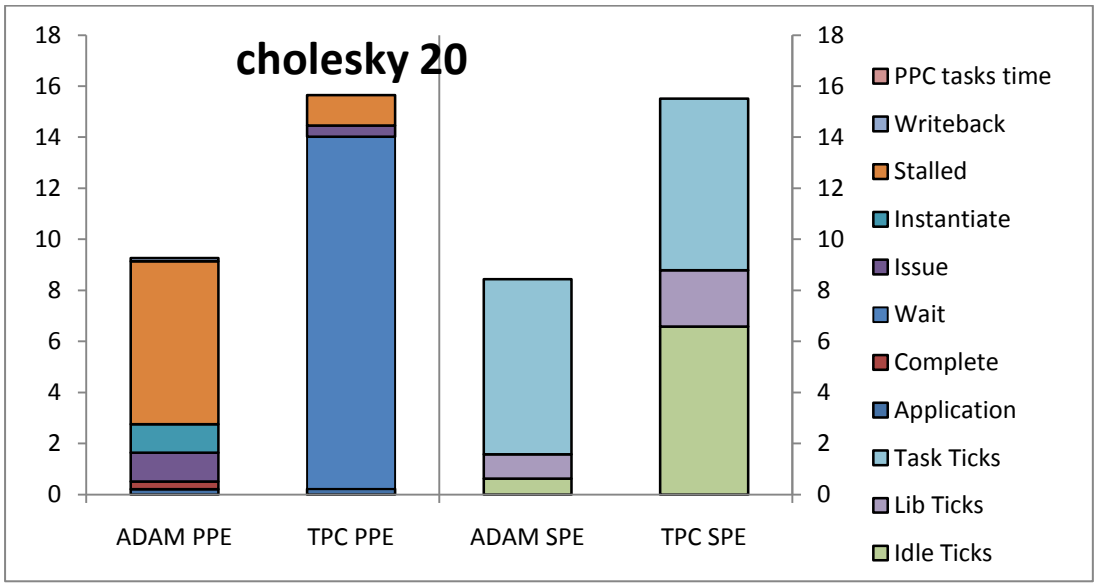

Figure 3.4.4—2 ADAM vs. TPC Cholesky 20x20. Cholesky execution Breakdown for the PPE and the SPEs for runs with 6 SPEs and for a 20x20 matrix of 64x64 blocks(Cholesky blocks) of single precision numbers.

3.4.5 Matmul

ADAM yields great performance in the matmul benchmark compared to the TPC [**3**] runtime, regardless of the dataset. This performance gain is due to the fact that dependencies cause the TPC [**3**] runtime to execute this benchmark sequentially. Therefore synchronization(wait) dominates in the execution time of the TPC [**3**] runs.

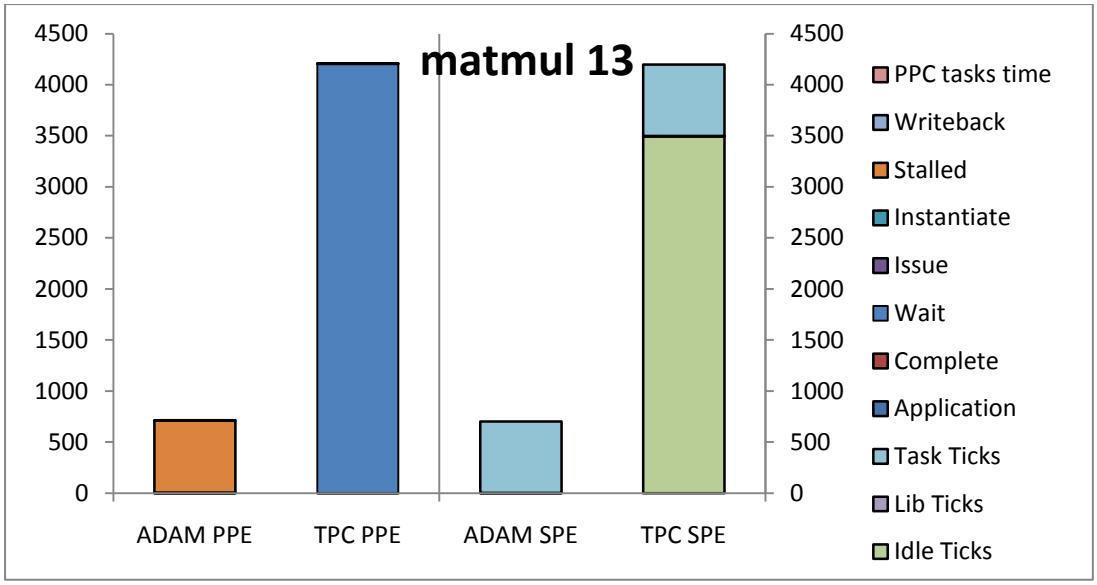

Figure 3.4.5—1 ADAM vs. TPC Matmul 13x13. Matmul execution Breakdown for the PPE and the SPEs for runs with 6 SPEs and for a 13x13 matrix of 64x64 blocks(Matmul blocks) of single precision numbers.

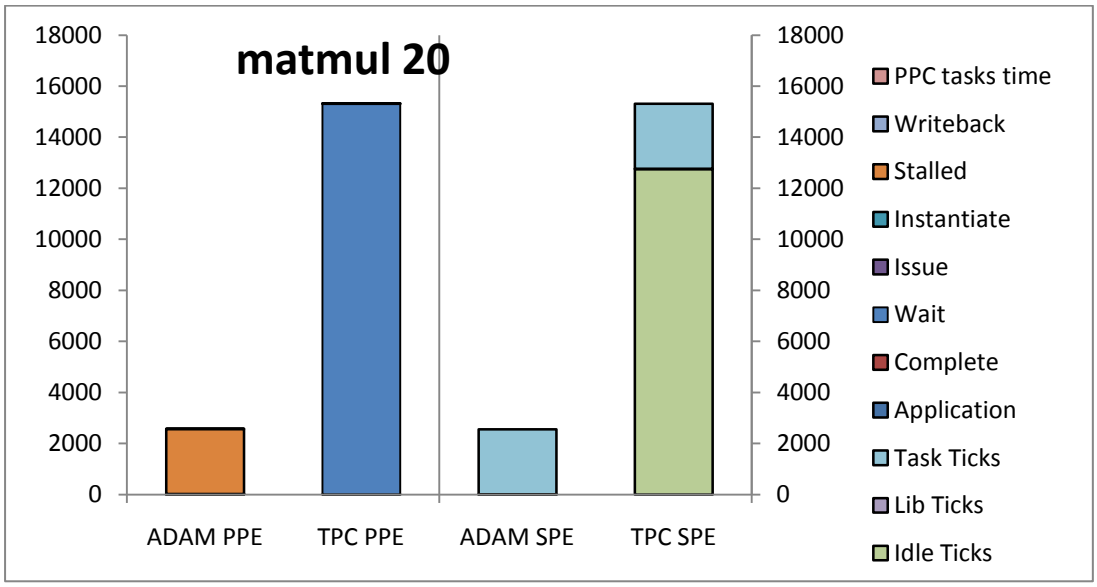

Figure 3.4.5—2 ADAM vs. TPC Matmul 20x20. Matmul execution Breakdown for the PPE and the SPEs for runs with 6 SPEs and for a 20x20 matrix of 64x64 blocks(Matmul blocks) of single precision numbers.

3.4.6 Jacobi

ADAM outperforms the TPC [**3**] in the Jacobi benchmark. The TPC [**3**] runtime cannot scale in the case of the Jacobi runtime due to the dependencies between tasks, among iterations for different blocks. ADAM outperforms the TPC [**3**] thanks to renaming, despite the fact that ADAMs overheads dominate its execution times.

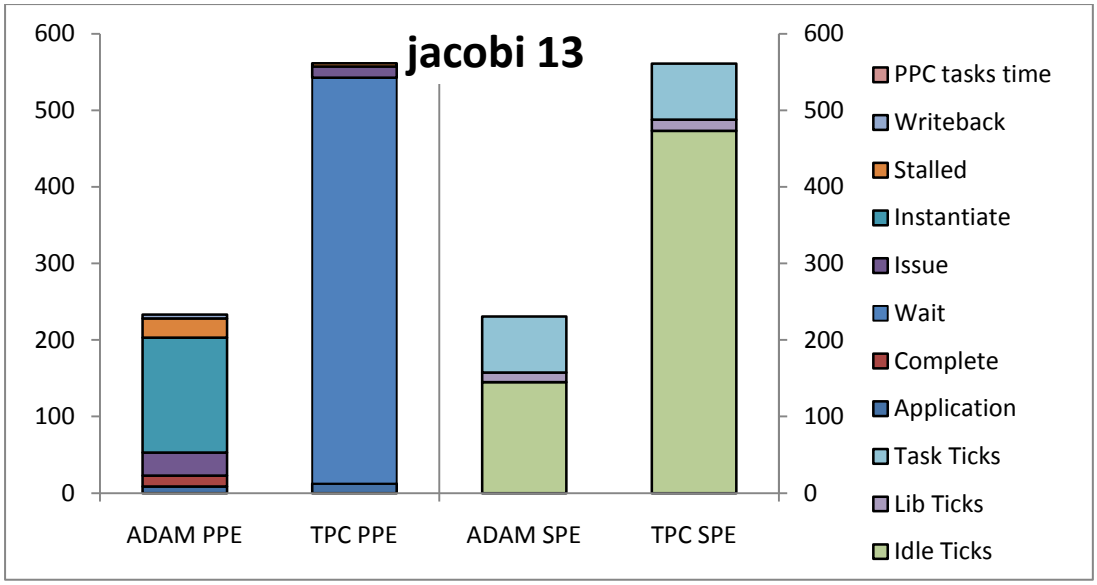

Figure 3.4.6—1 ADAM vs. TPC Jacobi 13. Jacobi execution Breakdown for the PPE and the SPEs for runs with 6 SPEs and for 13 iterations on a 32x32 matrix of 32x32 blocks(Matmul blocks) of single precision numbers.

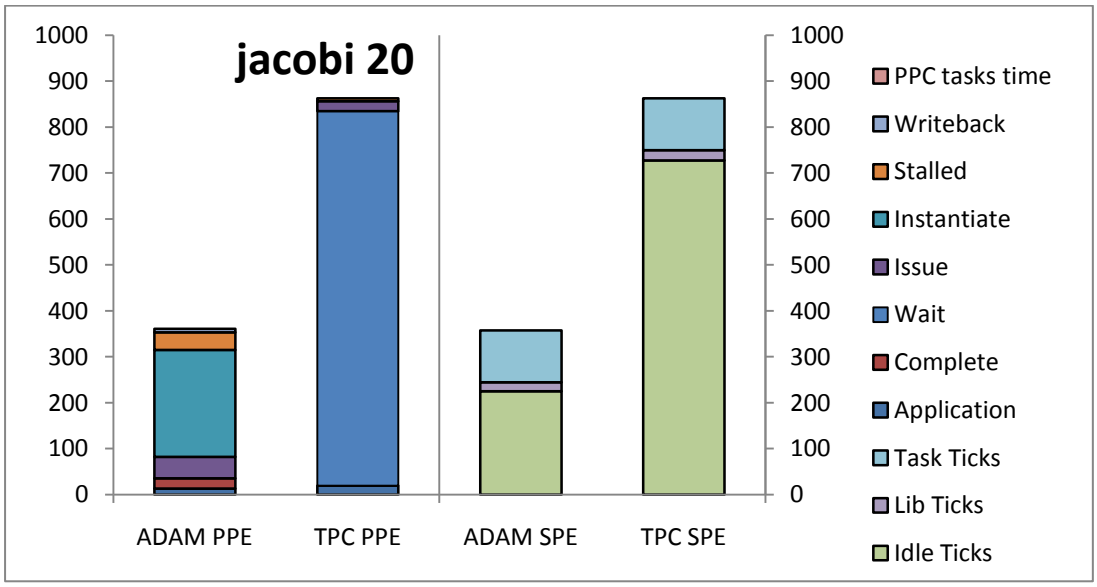

Figure 3.4.6—2 ADAM vs. TPC Jacobi 20. Jacobi execution Breakdown for the PPE and the SPEs for runs with 6 SPEs and for 20 iterations on a 32x32 matrix of 32x32 blocks(Matmul blocks) of single precision numbers.

3.5 Evaluating against Manual dependence Analysis

We now compare ADAM to a runtime in which the dependencies are explicitly expressed by the programmer(**Appendix A**). The programmer annotates dependencies among tasks using task-ids. In this runtime tasks follow the same scheduling path as in ADAM. The runtime does not perform renaming and because the dependence analysis is performed by the programmer the runtime overheads are lower than ADAM. In any case where data-renaming is not a factor Manual dependence analysis should perform better. It is, however, extremely strenuous for the programmer to properly and effectively express dependencies in the manual dependencies runtime, because in order to match ADAMs performance the programmer must express dependencies for every task instance and not generically based on task definitions. Furthermore the programmer must manage the dependence ids without introducing excessive overhead.

We compare the two runtimes in all of our benchmarks except for FFT because we cannot express dependencies involving strided arguments. The following graphs compare runs with 6 workers (SPEs). In the graphs the breakdowns for ADAM remain the same and the Breakdowns for this runtime are the following:

Manual dependence analysis PPE Breakdown

- **Instantiate**
  Instantiate is the time dedicated to creating a task. This time includes the overhead of the dependence analysis required for each argument of the task.
- **Complete**
  Complete is the time required to update the dependency graph for each completed task.
- **Issue**
  Issue is the time required for the master to send the task descriptors to the workers. The task descriptor is sent via remote stores.
- **Stalled**
  This is the portion of time in which the Master has tasks eligible for execution and polls the worker queues for an available slot.
- **Wait**
  This is the measured time of the Master blocking until workers complete their tasks. This corresponds to the barrier synchronization time. ADAM requires only one barrier at the end of the application.
- **Writeback**
  Writeback is the measured time of the Master copying renamed buffers back to the original addresses
- **PPC tasks time**

This is the time the Master spends executing Tasks

- **Application**
  This is the time dedicated to the running application excluding the aforementioned overheads and initialization.

Manual dependence analysis SPE Breakdown

- **Task Ticks**
  Task is the portion of time the SPE spends executing task code.
- **Lib Ticks**
  Lib is the portion of time the SPE spends executing library code.
- **Idle Ticks**
  Idle is the time spent by an SPE when it has no pending or executing tasks.

3.5.1 LU

Manual dependence analysis yields better performance in the runs with 8x8 block size(**Figure 3.5.1—1,Figure 3.5.1—4**). In the rest of the runs for the LU application, manual dependence analysis matches ADAM's performance. The introduced overheads by the manual dependence analysis runtime are significantly lower compared to ADAM's. This is why in the 8x8 runs where ADAMs performance is hindered by its overheads the manual dependence analysis outperforms ADAM.

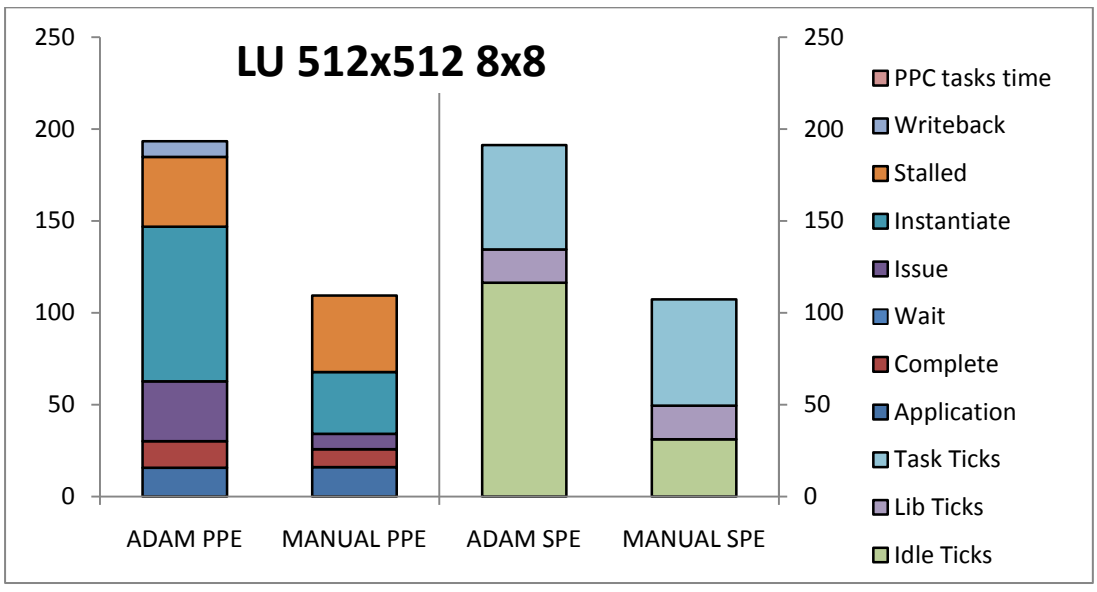

Figure 3.5.1—1 ADAM vs. Manual Dependence Analysis LU 512x512(8x8). LU execution Breakdown for the PPE and the SPEs for runs with 6 SPEs, for an array size of 512x512 with application block size 8x8.

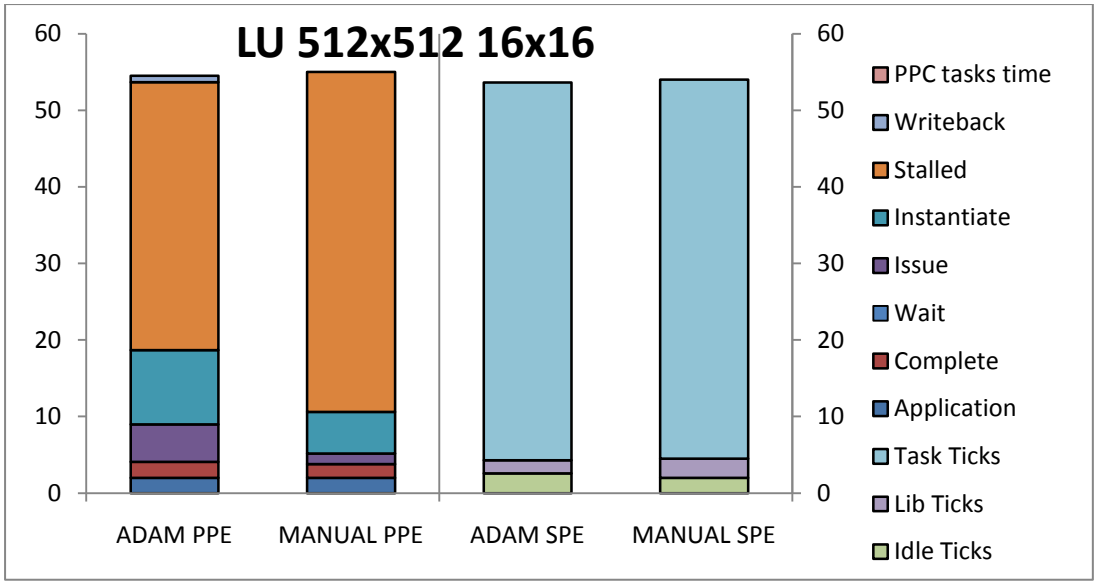

Figure 3.5.1—2 ADAM vs. Manual Dependence Analysis LU 512x512(16x16). LU execution Breakdown for the PPE and the SPEs for runs with 6 SPEs, for an array size of 512x512 with application block size 16x16.

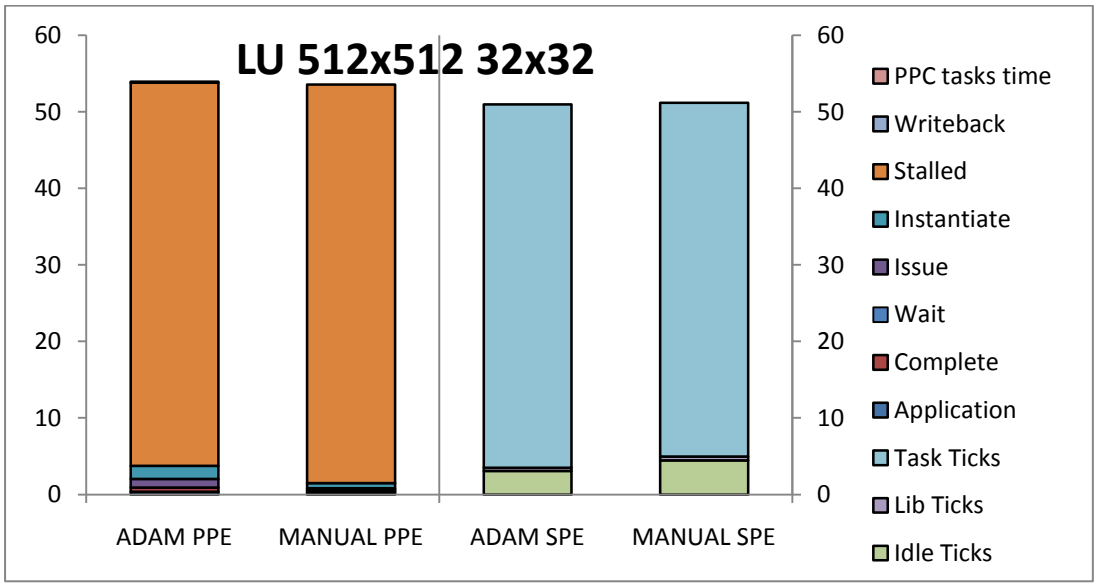

Figure 3.5.1—3 ADAM vs. Manual Dependence Analysis LU 512x512(8x8). LU execution Breakdown for the PPE and the SPEs for runs with 6 SPEs, for an array size of 512x512 with application block size 32x32.

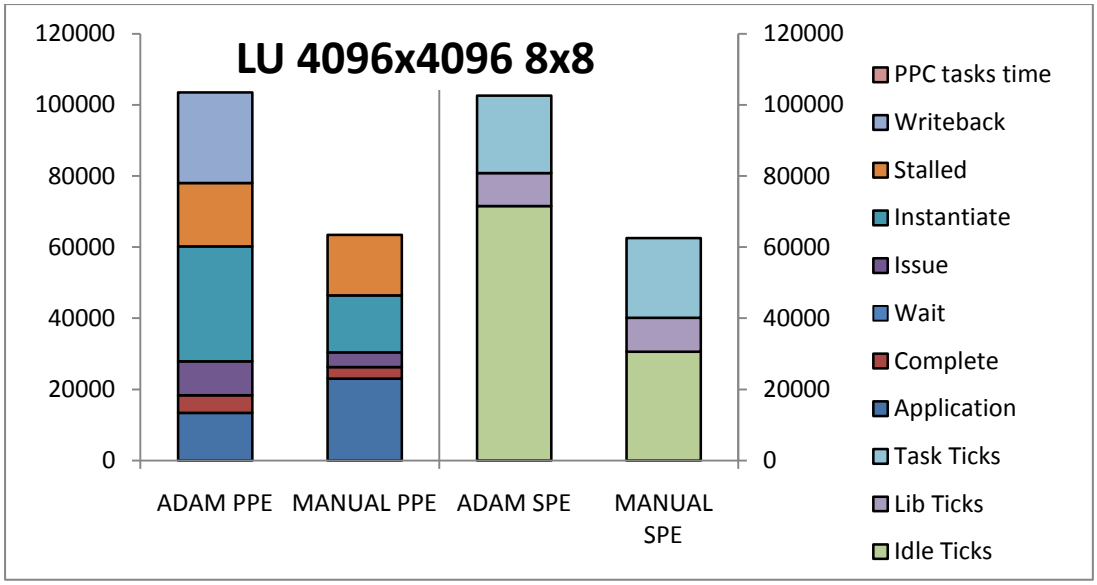

Figure 3.5.1—4 ADAM vs. Manual Dependence Analysis LU 4096x4096(8x8). LU execution Breakdown for the PPE and the SPEs for runs with 6 SPEs, for an array size of 4096x4096 with application block size 8x8.

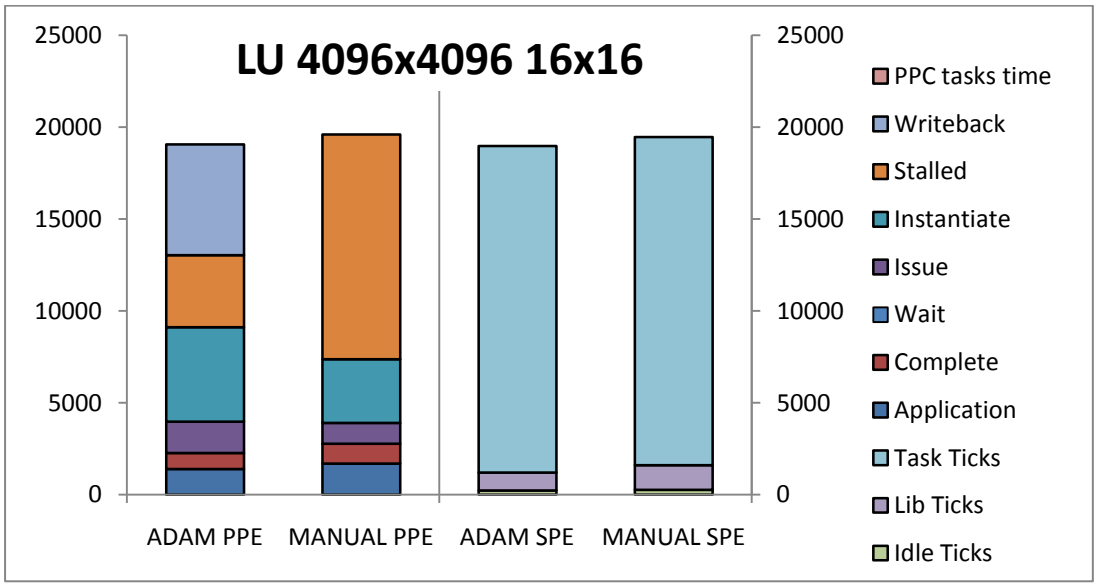

Figure 3.5.1—5 ADAM vs. Manual Dependence Analysis LU 4096x4096(16x16). LU execution Breakdown for the PPE and the SPEs for runs with 6 SPEs, for an array size of 4096x4096 with application block size 16x16.

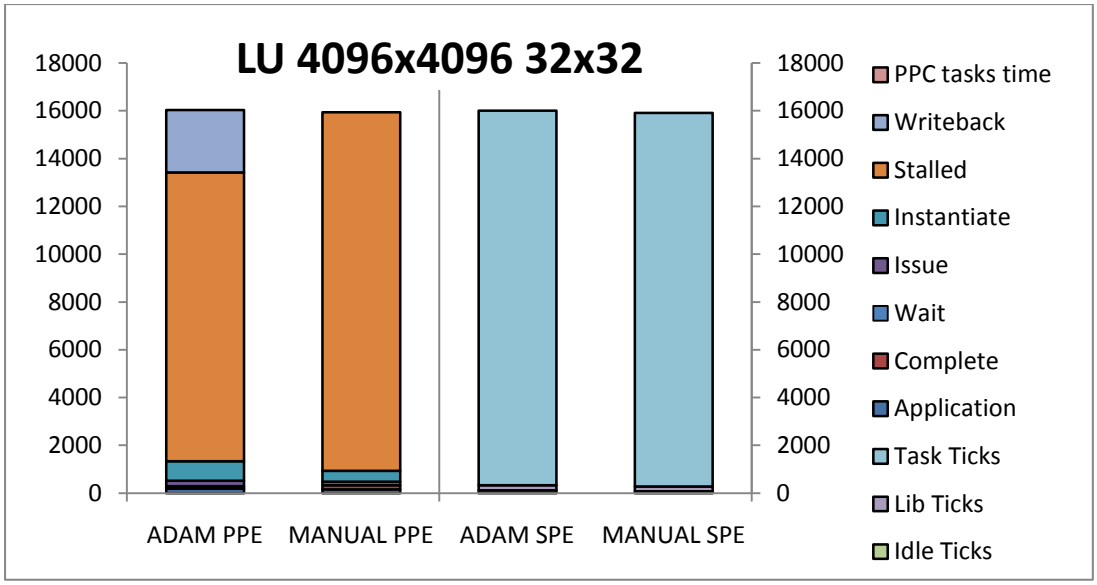

Figure 3.5.1—6 ADAM vs. Manual Dependence Analysis LU 4096x4096(32x32). LU execution Breakdown for the PPE and the SPEs for runs with 6 SPEs, for an array size of 4096x4096 with application block size 32x32.

3.5.2 Sequoia [4] kernels

Manual dependence analysis outperforms ADAM in all the runs concerning the Sequoia [**4**] micro-benchmarks. ADAMs performance in these benchmarks is bound by the introduced overheads due to the small size of the tasks. Manual dependence analysis however introduces extremely lower overheads and thus yields improved performance compared to ADAM in all cases(**Figure 3.5.2—1,Figure 3.5.2—2,Figure 3.5.2—3**).

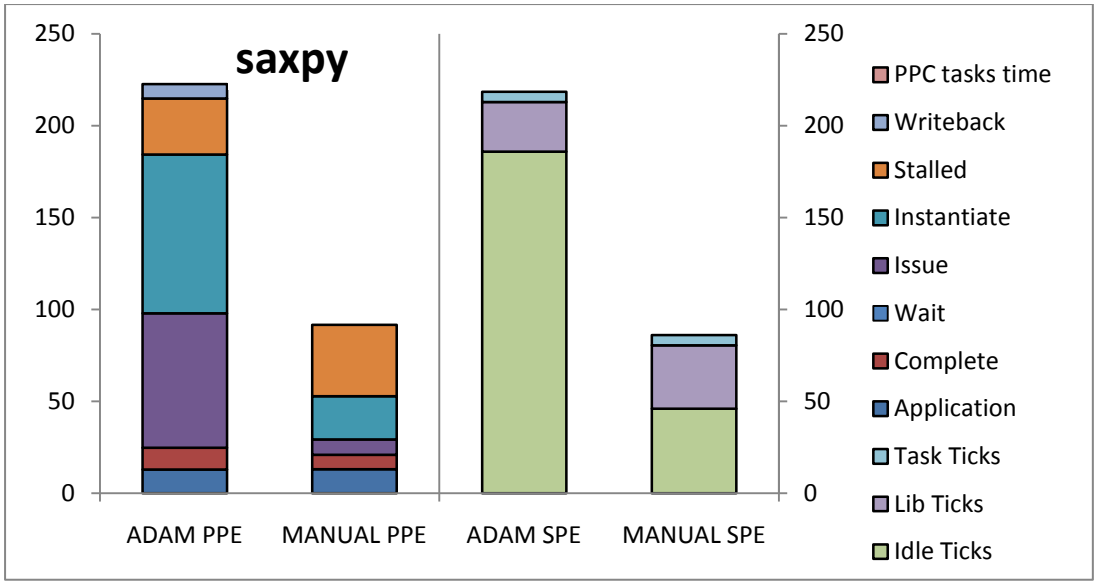

Figure 3.5.2—1 ADAM vs. Manual Dependence Analysis SAXPY. Saxpy execution Breakdown for the PPE and the SPEs for runs with 6 SPEs.

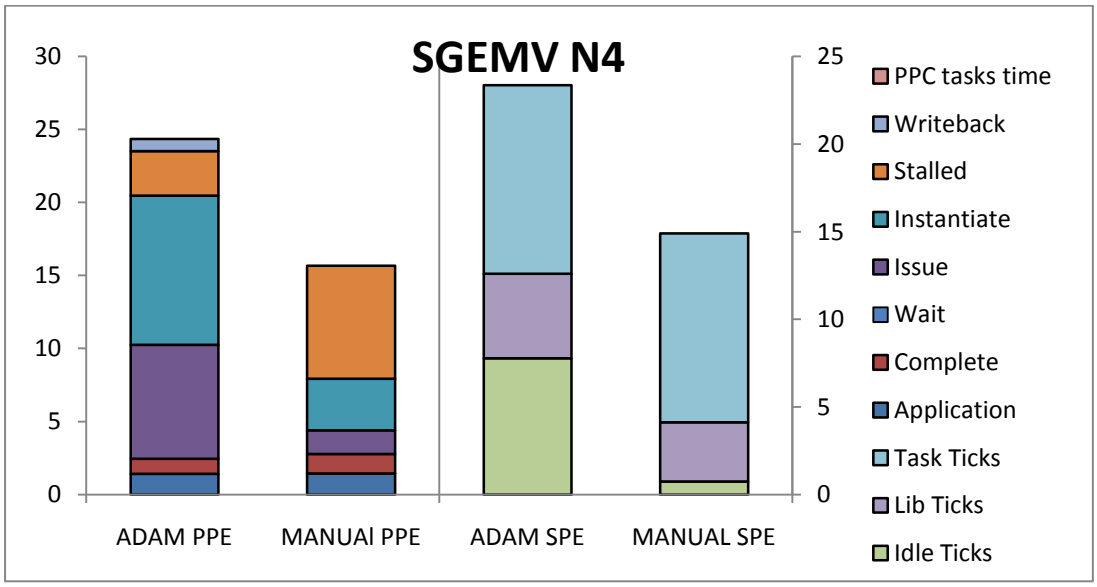

Figure 3.5.2—2 ADAM vs. Manual Dependence Analysis SGEMV N4. Saxpy execution Breakdown for the PPE and the SPEs with N=4 for runs with 6 SPEs.

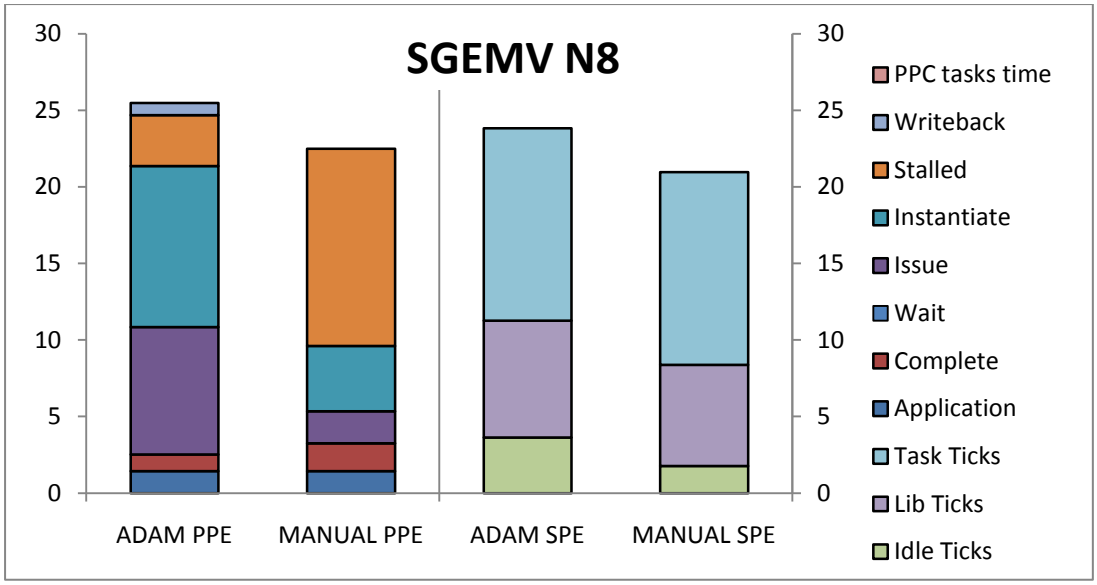

Figure 3.5.2—3 ADAM vs. Manual Dependence Analysis SGEMV N8. Saxpy execution Breakdown for the PPE and the SPEs with N=8 for runs with 6 SPEs.

3.5.3 Cholesky

Regarding the Cholesky benchmark the two runtimes achieve close performance. Manual dependence analysis yields lower overheads but ADAM performs better analysis, because the application presents a lot of dependencies and the user dependence analysis is not as effective as ADAMs. Further tuning of the application would produce better results on behalf of the manual dependence analysis runtime.

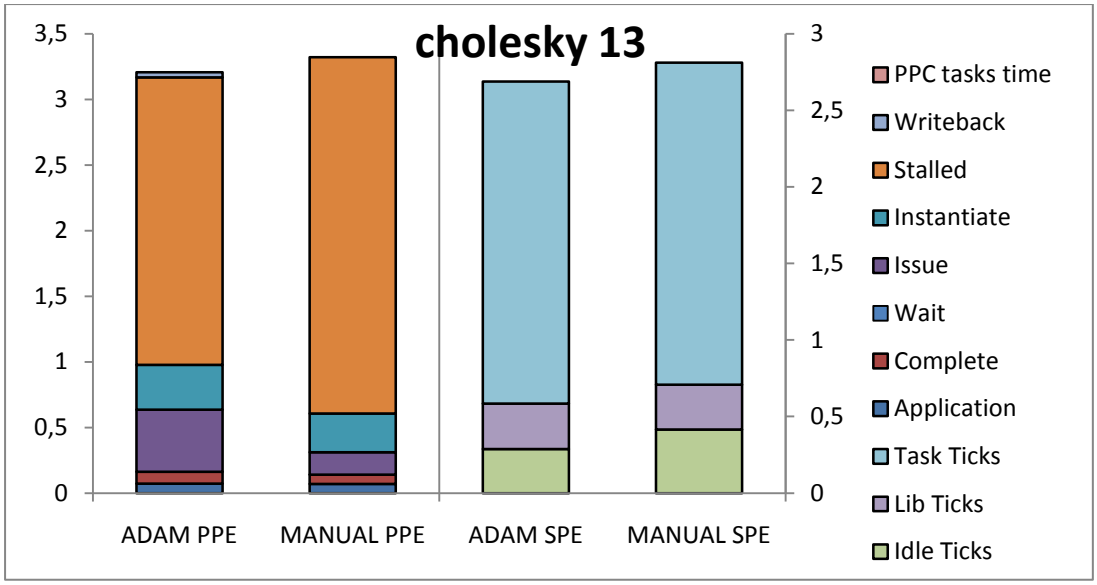

Figure 3.5.3—1 ADAM vs. Manual Dependence Analysis Cholesky 13x13. Cholesky execution Breakdown for the PPE and the SPEs for runs with 6 SPEs and for a 13x13 matrix of 64x64 blocks(Cholesky blocks) of single precision numbers.

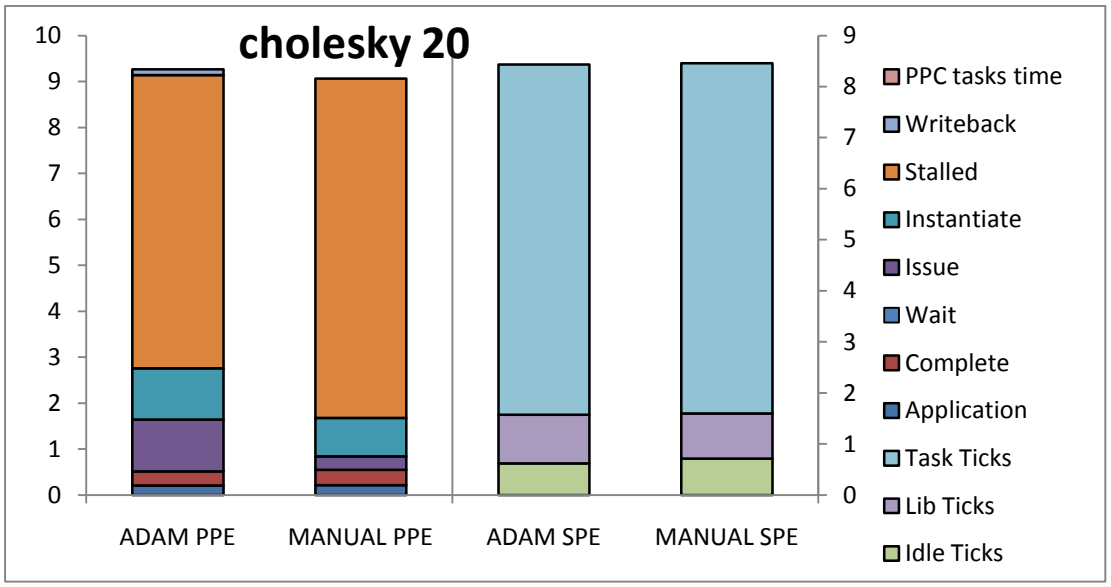

Figure 3.5.3—2 ADAM vs. Manual Dependence Analysis Cholesky 20x20. Cholesky execution Breakdown for the PPE and the SPEs for runs with 6 SPEs and for a 20x20 matrix of 64x64 blocks(Cholesky blocks) of single precision numbers.

3.5.4 Matmul

In case of the matmul benchmark both the runtimes perform identical. The overheads in this benchmark account for a very small portion of the execution time, therefore the manual dependence analysis lower overheads do not present any performance gain.

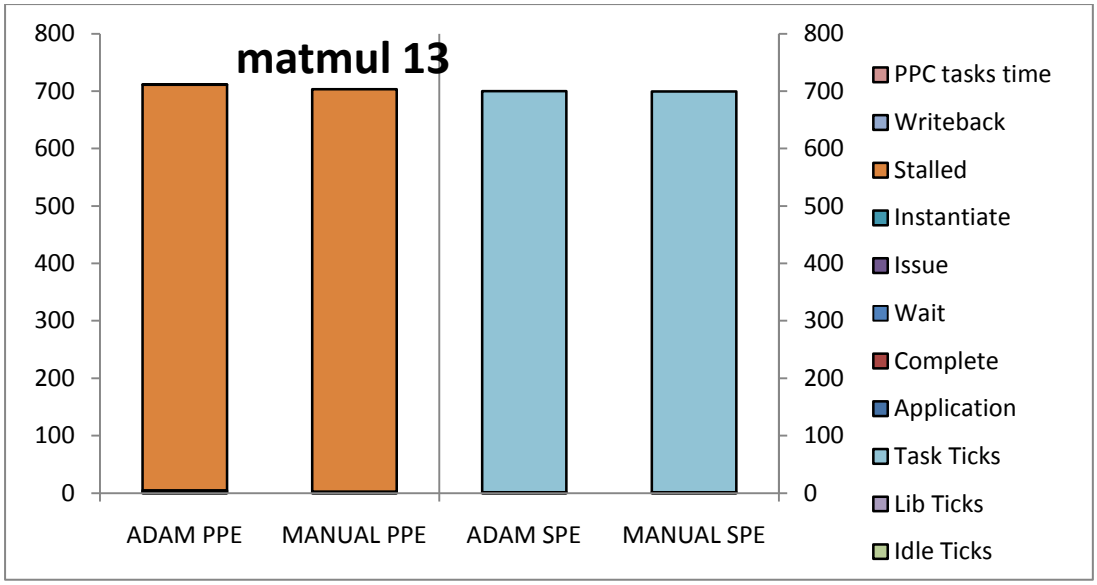

Figure 3.5.4—1 ADAM vs. Manual Dependence Analysis Matmul 13x13. Matmul execution Breakdown for the PPE and the SPEs for runs with 6 SPEs and for a 13x13 matrix of 64x64 blocks(Matmul blocks) of single precision numbers.

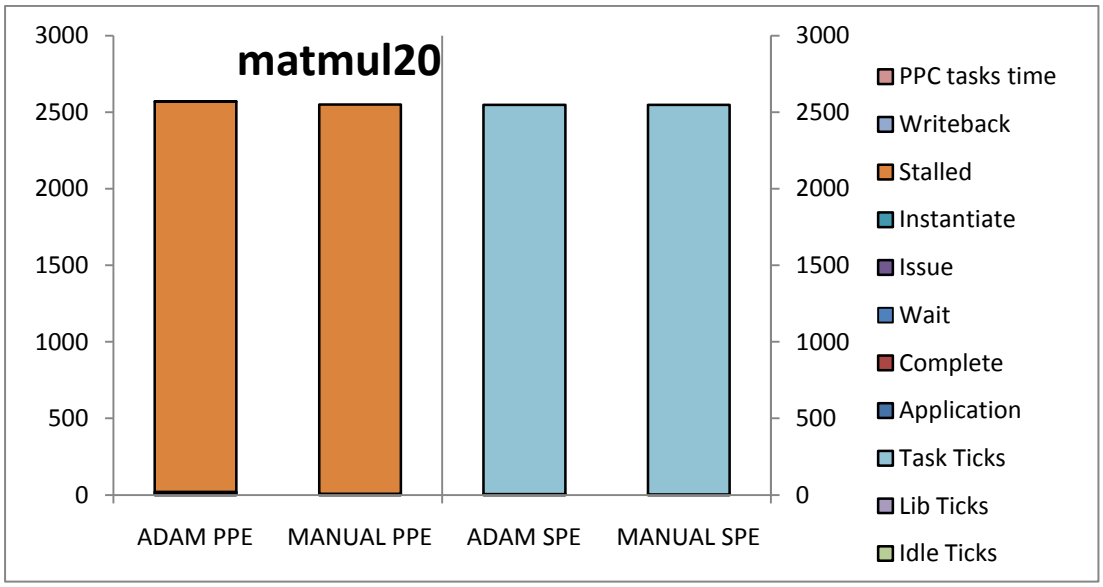

Figure 3.5.4—2 ADAM vs. Manual Dependence Analysis Matmul 20x20. Matmul execution Breakdown for the PPE and the SPEs for runs with 6 SPEs and for a 20x20 matrix of 64x64 blocks(Matmul blocks) of single precision numbers.

3.5.5 Jacobi

For the Jacobi benchmark ADAM performs significantly better than the manual dependence analysis runtime. Although ADAMs overheads are much larger than the manual dependence analysis runtime, ADAM extracts more parallelism through data-renaming. The

manual dependence analysis does not support data-renaming and therefore cannot export any parallelism thus executing the Jacobi sequentially.

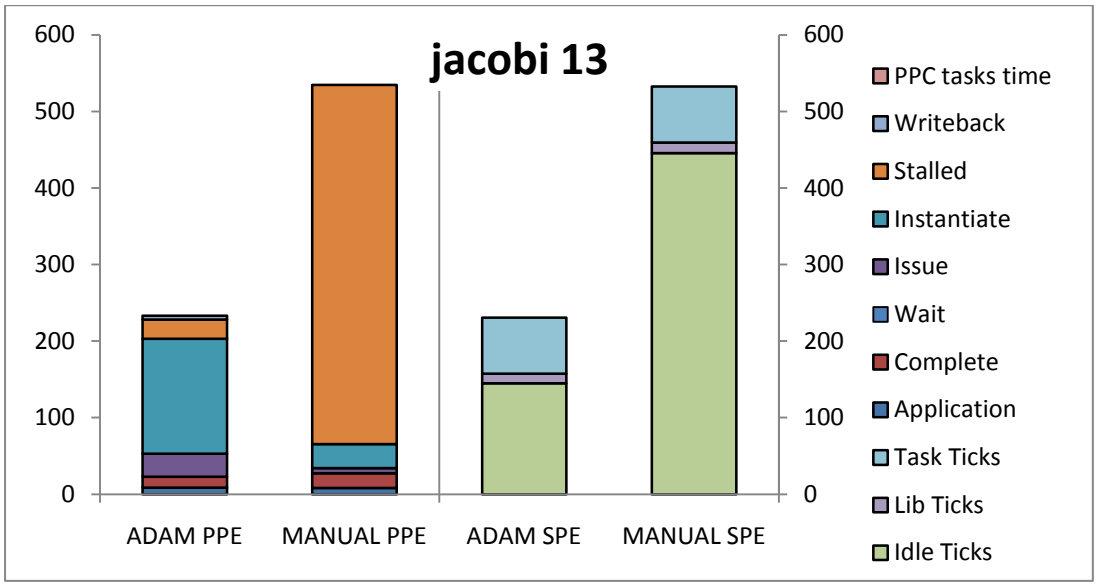

Figure 3.5.5—1 ADAM vs. Manual Dependence Analysis Jacobi 13. Jacobi execution Breakdown for the PPE and the SPEs for runs with 6 SPEs and for 13 iterations on a 32x32 matrix of 32x32 blocks(Matmul blocks) of single precision numbers.

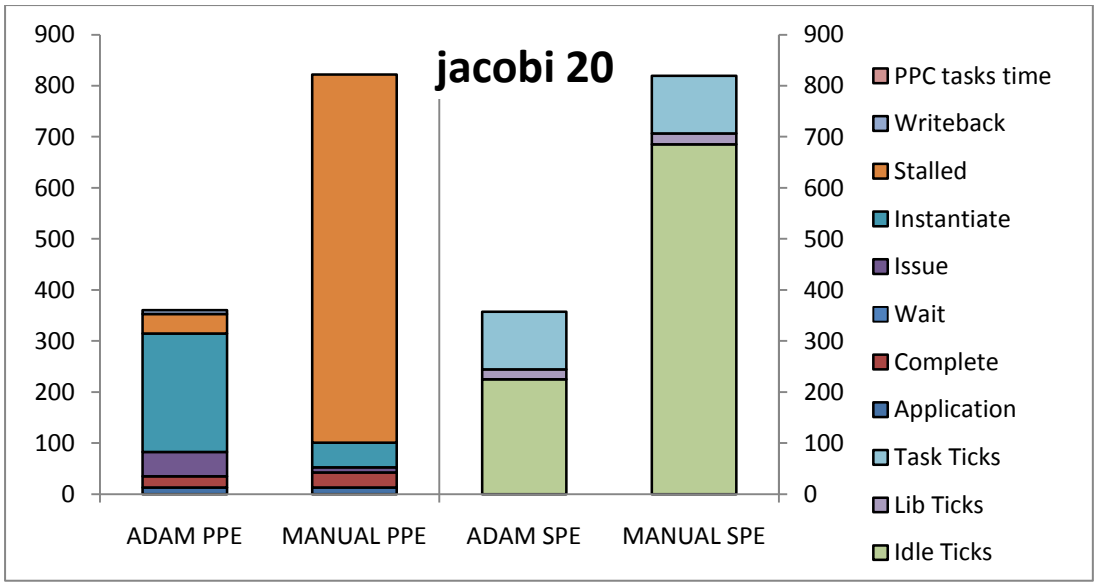

Figure 3.5.5—2 ADAM vs. Manual Dependence Analysis Jacobi 20. Jacobi execution Breakdown for the PPE and the SPEs for runs with 6 SPEs and for 20 iterations on a 32x32 matrix of 32x32 blocks(Matmul blocks) of single precision numbers.

4 Related Work

The introduction of new multi-core architectures that are limited from the existing programming models and languages has spawned a series of research efforts that aim on improving programmability and performance. In this section we briefly describe those efforts that, we feel, are related to our work. These efforts fall in the fields of programming models for the Cell [**1**] processor and software data-flow execution.

StreamIT [**10**] is a programming language for the streaming application domain. As a language it exposes streams as first-class objects thus allowing for compiler optimizations and improving programmer productivity. StreamIT [**10**] also serves as a common machine language for grid based architectures. The runtime model implemented in StreamIT [**10**] is based primarily on dataflow-execution, like ADAM. In StreamIT [**10**] however the data-flow model is produced from language expressions, while in ADAM the data-flow model is produced from the runtime analysis of task arguments, thus simplifying programming. Furthermore ADAM is not targeted to a specific application domain.

Sequoia [**4**] is a programming language that allows for task-based programs to take advantage of the memory-hierarchy. In order to do so, Sequoia [**4**] provides first-class objects for the memory hierarchy model and an abstract memory hierarchy model, so as to ensure portability. The mapping of tasks to the memory hierarchy is done explicitly by the programmer. Compared to ADAM, Sequoia [**4**] strives for data locality, while ADAM is locality unaware, but Sequoia [**4**] relies heavily on static program description and not on a data-flow runtime model. The locality optimizations of Sequoia can be easily implemented in ADAM using alternative task-to-core mapping schemes.

TBLAS [**11**] is a task-based runtime library intended for blocked linear algebra applications that performs dynamic task scheduling. TBLAS [**11**] is focused around improving the scalability of blocked linear algebra applications on distributed memory machines. In order to do so, TBLAS [**11**] proposes a decentralized runtime dependence analysis system with reduced communication overhead. ADAM on the other hand is centralized, a design influenced by heterogeneous processors that typically have one general-purpose core for control-intensive code and many special-purpose cores for compute-intensive code, and focuses on the efficiency of the runtime dependence analysis itself. Furthermore ADAM is not application specific.

OpenMP [**12**] is an API for shared memory parallel programming. OpenMP [**13**] supports both task and data parallelism. Tasks in OpenMP [**13**] are treated as separate work units of code without consideration for data-accesses. ADAM requires tasks to be treated as a unison with their data. Furthermore the task runtime model of OpenMP [**13**] does not support data-flow execution, instead it relies on programmer synchronization for proper execution. Ongoing research explores extensions of OpenMP to express data access attributes, similar to the in/out/inout attributes analyzed by ADAM, to implement dynamic dataflow execution of OpenMP tasks [**14**].

Cilk [**15**] is a programming language extension for the C programming language. Cilk [**15**] relies on the programmer to explicitly identify all the potential parallelism. A runtime system is responsible for exploiting all the annotated parallelism. Parallelism on Cilk [**15**] is based on the spawn-sync model that favors recursive parallelism. Cilk [**15**] also employes Work-stealing a technique that originates from Cilk [**16**] itself. In ADAM, the programmer identifies the potential parallelism by specifying tasks and then the runtime system discovers the actual parallelism. Furthermore ADAM is a pure-runtime solution with no language extensions required. ADAM though does not support work-stealing nor recursive code expression, although there is provision however for the support of these features in the SMP version of ADAM. Recent work [**17**] extended Cilk to include dataflow annotations, in order to express elegantly pipelined parallel computations. We plan to integrate this work with ADAM to accelerate runtime dependence analysis of Cilk codes.

CELLSS [**2**] is the originating runtime system for the STARSS runtime family [**18**]. CELLSS [**2**] uses the same task-based programming model as ADAM. It also employs runtime dependence analysis to export more parallelism through data-renaming and task re-ordering, and although in those aspects CELLSS [**2**] and ADAM are similar, the two runtimes differ vastly. CELLSS [**2**] focuses mainly on programmer ease and relies on the exported parallelism to compensate for the large runtime overheads. ADAM on the other hand focuses on high-performance alone and is designed to minimize overheads as well as to export as much parallelism as possible. ADAM and CELLSS [**2**] use different dependency models with CELLSS relying in a task-centric model while ADAM relies on a novel data-centric dependency model. On the scheduling part CELLSS [**2**] uses an additional thread called “helper thread” while in ADAM we chose a completely centralized scheduling method. To conclude, we believe that ADAM and CELLSS [**2**] are two very different solutions to the same problem. ADAM as shown in the Evaluation section outperforms CELLSS [**2**].

5 Conclusions and future work

In this work we propose a new way to enforce the data-flow execution model on task-based models. We also propose a new detection algorithm with O(1) cost per analyzed item and the ability to detect overlapping dependencies. We design and implement ADAM(Accelerated Dependence Analysis for Multi-cores), a runtime that adheres to our proposal, for the CELL [**1**] processor and for SMP x86 architectures. We evaluate ADAM for the CELL [**1**] processor and compared its performance to that of the TPC [**3**] runtime, of the CELLSS [**2**] runtime and of a manual dependence analysis runtime we have developed.

ADAM yields better performance than the CELLSS [**2**] runtime due to much lower overheads and a more aggressive renaming policy. ADAM is 2,77 times faster for the Cholesky benchmark with 13x13 dataset, 2,27 times faster for the Jacobi benchmark with 13 iterations and 7% faster in case of the Matmul benchmark.

Compared to the TPC [**3**] runtime ADAMs performance is worse only in cases where the task size do not compensate for the introduced overheads. There are also cases, like Jacobi and Matmul, where the TPC [**3**] runtime cannot express parallelism in an application because of the dependencies among tasks. In these cases ADAM outperforms the TPC [**3**]runtime.

The manual dependence runtime introduces lower overheads but does not perform data-renaming. Therefore whenever data-renaming does present a significant impact on performance, manual Dependence analysis always performs better. In our evaluated cases ADAM outperforms the manual dependence analysis only for the Jacobi benchmark. Manual dependence analysis however is extremely strenuous for the programmer and performance depends on programmer skills and the in-depth knowledge of the dependencies of the running application.

Furthermore ADAM's overall performance can be tuned through the parameters, analysis block size and task window size. Analysis block size is the granularity ADAM performs analysis with. The task window size is the total number of tasks issued allowed at any point time, and can trade between parallelism and analysis overhead. ADAM's performance can also be improved through the supported feature of PPE tasks, where ADAM allows the master to execute tasks.

In future work we intend to combine ADAMs dynamic dependence analysis with static dependence analysis. In later versions of ADAM the programmer will also be able to define Analysis block sizes in 2 dimensions, in order to accommodate analysis for strided arguments. Future work also includes exploring design alternatives for renaming that trade off memory space, with execution time. We also intend to explore the use of ADAM as a framework for the creation of parallel programming tools for other purposes, that may include deterministic replay, deterministic execution of parallel programs, debugging and feedback on locality and parallelism.

ADAM is a runtime system that successfully exports parallelism from applications. ADAM's major disadvantage is the introduced overhead. If the introduced overhead is tolerable then ADAM should be used. Finally we believe that dependence analysis is an important factor of application parallelization, and that its effect is always beneficial both in programmability and performance. The main challenge in dependence analysis between runtime and compile time dependence analysis is the trade-off of overhead versus effectiveness, and between automatic and manual is the trade-off of overhead versus programmability.

Appendix

A. Manual Dependence Analysis

In this section we describe the architecture of the manual dependence analysis runtime we designed and implemented. This is the runtime we compare ADAM with, in the evaluation section. The precondition, for this runtime system, is that all dependencies should be explicitly expressed by the programmer. The starting point for this runtime was ADAM and because of that the scheduler component and the workers are of the same design. For the parts of this runtime, whose description is omitted consider them identical to ADAM's.

The desired behavior for this runtime is to be able to track dependencies based on unique task-ids, provided by the programmer. This means that for every given id we must assert that (i) the id is unique, that (ii) the id corresponds to a task issued within the current task window, and that (iii) this task is not completed. The programmer cannot respect this assertions when providing the dependence ids, because they are subject to runtime conditions.

First of all we ensure uniqueness among task-ids. For every created task the runtime returns a unique task-id to the user in the form of an integer number. The runtime maintains internally a Lamport clock [**9**] called "epoch", which is increment every time a new task is created. The unique-id for every task is the current value of "epoch". Therefore the unique-id is actually a number representing the issue order of each task (i.e. Nth issued task will have task-id N).

We also need to distinguish ids that belong to the current task window. For each task the runtime maintains the full task description consisting of a function id and a triplet {address, size, type} for each argument. In addition to this information the task maintains a counter, named input_dependencies_counter, that holds the number of pending unresolved dependencies for this task, a stack that holds all the tasks that depend on it and a field called epoch that holds the task's "epoch" value. Using the "epoch" value as an normailzed index we place each task upon creation on an array called TimeLine. Let us assume first that the TimeLine is of size that matches the task window. The TimeLine at any given point contains the task's within our Task window. For a given task-id say '**I**' we index the TimeLine

and retrieve task '**T**'(*T = TimeLine[I % TimeLineSize]*). If the value of '**I**' matches the value of '**T**' 's epoch (*T->epoch == I*)then this ID is considered of the current task window. Because the Task window is a sliding window with a slide step defined by the programmer(task window threshold), a task with epoch N in its lifetime may co-exist with tasks from the epoch range (N – "Task Window" , N + "Task Window"), therefore it is not sufficient for the TimeLine to be of the same size as Task-window. To ensure correctness we always set the TimeLine to the size of two Task Windows, although it can be refined to a smaller value depending on the task window threshold.

We use the timing model described in the previous paragraph to also distinguish between completed and alive tasks. We consider that the runtime systems internal Lamport clock [**9**] called "Epoch" starts counting from the value "1" and not "0". Because the dependence analysis part of the runtime is indifferent of executed tasks, we consider that all executed tasks occurred in the past, at time denoted "0". Whenever a task is completed we simply set its epoch value to "0". This strengthens the timing model so that for a given task-id say '**I**', If the value of '**I**' matches the value of '**T**' 's (*T = TimeLine[I % TimeLineSize]*) epoch (*T->epoch == I*)then this ID is considered both within the task-window and not completed. For programming ease we also allow the programmer the use of the value "0" as a no-dependencies id.

The dependence checking is described in the following pseudo code:

```
//we check for task-id ID
//"This" is the currently created task

If( ID!=0 ){
      T = TimeLine[ID % TimeLineSize];
      If( T->epoch == ID){
            Push(T->stack, This);
            This->input_dependencies_counter++;
      }
```

Whenever a task a task is created, for every valid dependence we detect the input_dependencies_counter is incremented by one and the current task is pushed onto the stack of the task it depends on. If the input_dependencies_counter is equal to 0 after the dependence detection step then this task is considered eligible for execution and it is pushed to the ready_queue. Whenever a task has completed execution we pop the tasks from its stack. For each task we pop we decrement the input_dependencies_counter by one.

After the decrement if the input_dependencies _counter is equal to 0 then this task is considered eligible for execution and it is pushed to the ready_queue.

6 References

[1] J. A. Kahle, et al., "Introduction to the Cell multiprocessor," *IBM Journal of Research and Development*, vol. 49, no. 4, pp. 589-604, 2005.

[2] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "CellSs: a Programming Model for the Cell BE Architecture," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.

[3] G. Tzenakis, et al., "Tagged Procedure Calls (TPC): Efficient Runtime Support for Task-Based Parallelism on the Cell Processor," in *Proc. of the 2010 International conference on High-Performance Embedded Architectures and Compilers (HiPEAC)*, vol. 5952, 2010, pp. 307-321.

[4] K. Fatahalian, et al., "Sequoia: programming the memory hierarchy," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.

[5] D. A. Ilitzky, J. D. Hoffman, A. Chun, and B. P. Esparza, "Architecture of the Scalable Communications Core's Network on Chip," *IEEE Micro*, vol. 27, no. 5, pp. 62-74, Sep. 2007.

[6] R. M. Karp and R. E. Miller, "Properties of a Model for Parallel Computations: Determinancy, Termination, Queueing," *SIAM Journal on Applied Mathematics*, vol. 14, no. 6, pp. 1390-1411, Nov. 1966.

[7] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," *SIGARCH Comput. Archit. News*, vol. 3, no. 4, pp. 126-132, Dec. 1974.

[8] A. Rico, A. Ramirez, and M. Valero, "Available task-level parallelism on the Cell BE," *Sci. Program.*, vol. 17, no. 1-2, pp. 59-76, Jan. 2009.

[9] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM* , vol. 21, no. 7, pp. 558-565, Jul. 1978.

[10] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *Proceedings of the 11th International Conference on Compiler Construction*, 2002, pp. 179-196.

[11] F. Song, A. YarKhan, and J. Dongarra, "Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009, pp. 19:1--

19:11.

[12] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE Computational Science & Engineering*, vol. 5, no. 1, pp. 46-55, 1998.

[13] E. Ayguade, et al., "The Design of OpenMP Tasks," *IEEE_J_PDS*, vol. 20, no. 3, pp. 404-418, 2009.

[14] ENCORE: ENabling technologies for a programmable many-CORE . [Online]. http://www.encore-project.eu/

[15] R. D. Blumofe, et al., "Cilk: an efficient multithreaded runtime system," in *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, 1995, pp. 207-216.

[16] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, 1998, pp. 212-223.

[17] H. Vandierendonck, P. Pratikakis, and D. S. Nikolopoulos, "Parallel Programming of General-Purpose Programs using Task-Based Programming Models.," in *Proc. of the 3rd USENIX Workshop on Hot Topics in Parallelism (HotPar)*, Berkeley, California, May 2011.

[18] J. labarta, "StarSs: A Programming Model for the Multicore Era," in *PRACE Workshop "New Languages & Future Technology Prototypes"*, Leibniz Supercomputing Centre in Garching (Germany), March 2010.