Using Chip Multithreading to Speed Up Scenario-Based Design Space Exploration: A Case Study

Peter van Stralen Andy D. Pimentel
Institute for Informatics, University of Amsterdam
{p.vanstralen, a.d.pimentel}@uva.nl

ABSTRACT

Early design space exploration (DSE) is a key element of system-level design of complex embedded systems, helping designers to make design decisions during the early design phases. For early DSE, where the design space is vast, it is crucial that the exploration process is as efficient as possible. In this paper, we describe the implementation of our scenario-based DSE framework on a Chip Multithreading platform, namely the SPARC T3-4 server, and study its performance behavior in detail.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Performance Attributes

General Terms

Design, Measurement, Performance

Keywords

Scenario-based design space exploration, Chip multithreading, Implementation study

1. INTRODUCTION

A significant amount of research has been performed on system-level Design Space Exploration (DSE) for Multi-Processor System-on-Chips (MPSoCs)s (e.g. [9, 5, 12, 11, 10, 6, 13]) during the last two decades. The majority of this work is focused on the analysis of MPSoC architectures under a single, static application workload. The current trend, however, is that application workloads executing on embedded systems become more and more dynamic.

Recently, we have introduced the scenario-based DSE environment [14] that exploits workload scenarios [4] to model both the dynamism between and within the applications that are mapped onto a MPSoC. As a basis for scenario-based DSE, a scenario-aware version of our high level simulation framework Sesame [15, 14, 17] is used. In this scenario-aware Sesame (as illustrated in Figure 1), the concept of separation of concerns is used, resulting in separate models for the application(s), the architecture and the mapping. In the application model the functional behavior of the application workload is described using intra- and inter-application scenarios. Intra-application scenarios describe the dynamic behavior within an application, whereas the inter-application scenarios describe the dynamic behavior between multiple applications (i.e., which applications can run concurrently). The structure of the applications themselves are described using Kahn process networks [7]. In Figure 1, the example application workload consists of two applications, a MP3 decoder and a Video decoder. The architecture model describes the non-functional behavior (e.g., execution cycle times, power consumption, etc.) of the MPSoC design. To connect the application model and the architecture model, the mapping layer maps both the processes and communication channels in the applications onto a component in the architecture model. To this end, the mapping layer deploys a trace-driven simulation approach to co-simulate the application model (generating event traces) and the architecture model (consuming the event traces) [11].

A complicating factor of embedded system design, however, is that there is an exponential number of potential mappings. That is why a search algorithm is required to efficiently explore the design (i.e., mapping) space. Therefore, our scenario-based DSE framework aims at an efficient search for (sub-)optimal mappings of embedded systems with dynamic multi-application workloads. As shown in Figure 2, the framework consists of two components: the design explorer and the subset selector. The design explorer searches for optimal MPSoC mappings. For this purpose, a genetic algorithm [8, 3] is used that applies natural evolution on a

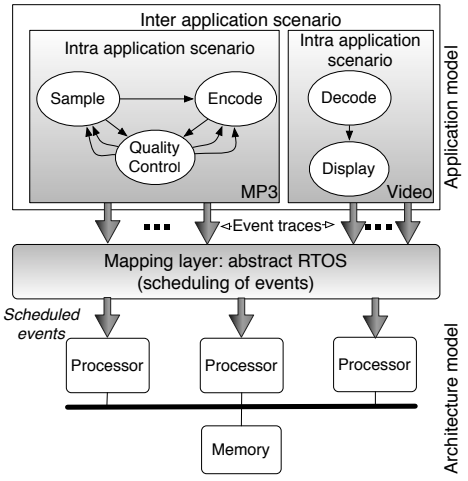


Figure 1: High level scenario-based MPSoC simulation.

RAPIDO'14 January 2014, Vienna, Austria.

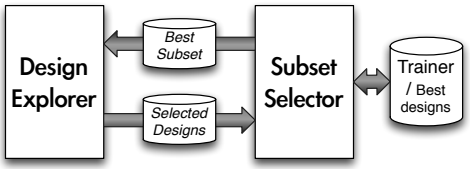


Figure 2: The framework for scenario-based DSE.

population of mappings to identify high quality mappings. This quality is determined by simulating each mapping in the population. Although a Sesame simulation typically takes less than a second, there are many mappings that need to be evaluated. On top of that, each mapping needs to be evaluated for multiple application scenarios. To speed up the evaluation of a single mapping, a representative subset of application scenarios is used for the evaluation instead of using all possible application scenarios. This representative subset of scenarios is identified by the subset selector. Based on a set of training mappings (that is based on selected designs from the design explorer), the subset selector dynamically selects a representative scenario subset. Since the representativeness of a subset of scenarios is dependent on the designs explored in the design explorer, both the design explorer and the subset selector are executing simultaneously [16, 17].

To efficiently perform scenario-based DSE, we have implemented it on a Chip Multithreading system, namely the SPARC T3-4 server, and studied its performance behavior. With its 512 hardware threads, the SPARC T3-4 server should be a suitable target for for the highly parallel scenario-based DSE. During the search process in scenario-based DSE, a number of worker threads are used to perform Sesame simulations in parallel. Over time, each worker thread fires many processes to perform simulation jobs that can be executed on the hardware threads of the SPARC T3-4.

In the remaining sections of this paper, we first provide a more detailed overview of the the SPARC T3-4 server. Next, Section 3 describes the unoptimized implementation of our scenario-based DSE. The following section (Section 4) shows how the scenario-based DSE has been profiled to optimize the final design of the scenario-based DSE (Section 5). After that, in Section 6, a range of experiments show both the performance and the bottlenecks of the scenario-based DSE framework. Finally, Section 7 concludes the paper.

2. THE SPARC T3-4

The SPARC T3 processor [1] tries to minimize the idle time of the processor by using Chip Multithreading (CMT). CMT is a combination of Chip Multiprocessor (CMP) and Fine-Grained Multithreading (FG-MT). With FG-MT, a processor core is capable of quickly switching between active threads. As a result, memory latency of a thread can be hidden by executing another active thread. In this way, the processor core tries to do active work in each cycle. A SPARC T3 processor has up to sixteen cores, each supporting eight hardware threads per core. In our case, a SPARC T3-4 server is used. This server is shown schematically in Figure 3. As the name suggests, the SPARC T3-4 has four SPARC T3 processors running at 1.65GHz. This gives a total of 64 T3 cores and a total number of 512 hardware threads.

Within a T3 core there are two execution units, one per a set of four hardware threads. Additionally, a single floating point / graphics unit (FGU) is present for all the eight hardware threads. Level 1 caches are present in each individual core. For instructions a 16KB instruction cache is available and data can be stored in a 8KB data

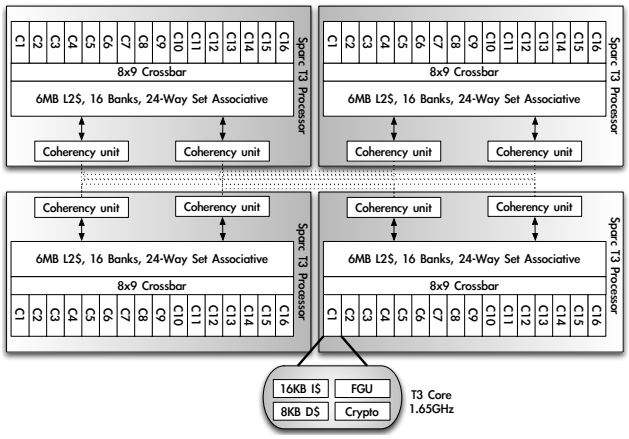


Figure 3: A schematic view of the SPARC T3-4 processor.

cache. The level 2 cache is banked and shared between all the cores on a single SPARC T3 processor using a crossbar switch. The cores in the SPARC T3 processor are kept consistent using coherency units. Finally, each T3 core has its own memory management unit for virtual memory management. An instruction Translation Lookaside Buffer (TLB) of 64 entries is present and the data TLB has 128 entries.

For comparing the performance of the scenario-based DSE on the SPARC T3-4, we have also executed the original implementation of our scenario-based DSE on a Sun Fire X4440 [2] compute server with four quad-core AMD Opteron 8356 processors running at 2.3GHz.

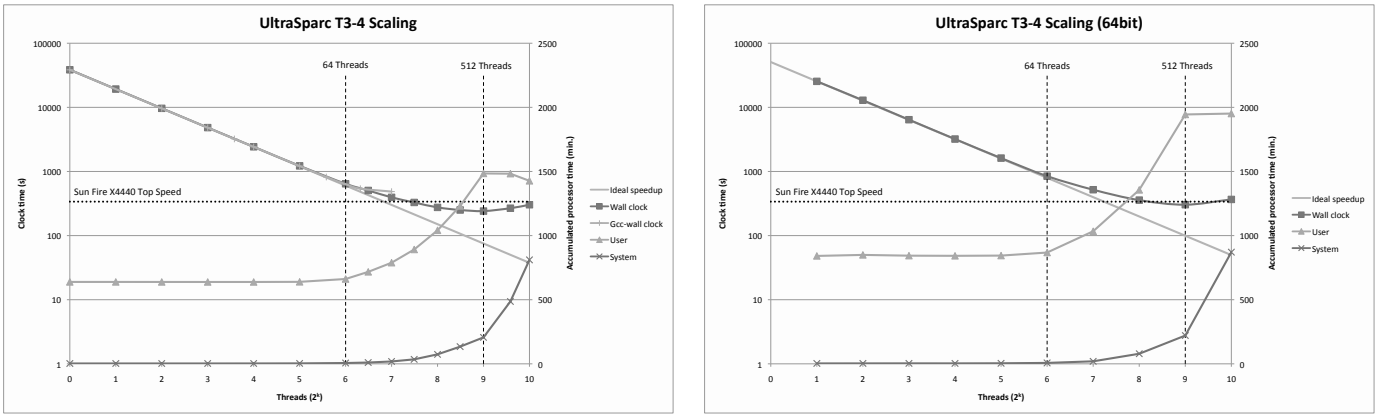
3. UNOPTIMIZED DSE IMPLEMENTATION

First, we have ported the scenario-based DSE to the SPARC T3-4 without any optimization. As the SPARC T3 processor supports 64 bits, we had the possibility to compile for both 32 and 64 bits. In order to see which option was best, we compared the performance scaling of both options. In Figure 4, the performance scaling is shown of the unmodified scenario-based DSE with default compilation flags[1]. During this experiment, the size of the DSE workload has been kept fixed while the number of worker threads is varied. First, we investigated the total wall clock time of the DSE experiment. The logarithmic horizontal axis shows the number of threads that are used to simultaneously process the simulations, whereas the left vertical axis shows the wall clock time, also using a logarithmic scale. As a consequence, the ideal speedup manifests itself with a straight line.

With 32-bit compilation, a comparison is made between the compiled code of gcc and compiled code of the Sun cc compiler. Up to 128 threads, the performance of the Sun cc compiler and the gcc compiler is similar and, therefore, we have decided to use the Sun cc compiler for all the experiments. For this compiler, the speedup of the code is completely linear until 64 threads. With more than 64 threads the speedup quickly decreases. At the optimal point of 512 threads, the Oracle SPARC T3-4 is 29 percent faster than our Sun Fire X4440 server.

To identify the cause of the decrease in speedup after 64 threads, the user and system time are also plotted in Figure 4. For these graphs, the right vertical axis shows the accumulated processor time in minutes. Since the amount of work is constant, the expected

[1]Optimization `-O3` for gcc and `-xO5` for the Sun cc compiler. Other flags have been tried, but did not have any significant effect.



(a) (b)

Figure 4: The performance scaling of Sesame on the Oracle SPARC T3-4 for (a) the initial 32 and (b) the 64 bit compilation.

behavior is that the accumulated user time also remains constant. This is indeed the case when the number of threads is between 1 and 64. After this point, however, the user time starts to increase. The same is true for the system time: due to the increased number of threads, the complexity of the coordinating tasks of the operating system starts to increase. This can be clearly seen when the system is overloaded with more than 512 threads.

For the user time, however, the increase is not as one would expect. The most plausible explanation is that the increase is related to the number of physical cores. There are 64 physical cores on the Oracle SPARC T3-4. When the number of threads is lower or equal than the number of physical cores, then the speedup is linear. With more than 64 threads, the hardware threads should lead to a further improvement of the performance. Still, this improvement is not linear anymore. Although hardware threads in a core allow for some parallel execution due to the two execution units, a large part of the functionality is shared with the other hardware threads on the physical core. To gain more insight in the underlying performance behaviour of the T3-based scenario-based DSE, we have applied a detailed profiling study that will be described in the following two sections.

Finally, the comparison between the 32-bit (Figure 4(a)) and the 64-bit implementation (Figure 4(b)) shows that the 32-bit version on the SPARC is superior to the 64-bit one. In our reference system, the x86 based Sun Fire X4440, 64-bit compilation is used, which clearly yields a faster implementation than when 32 bit compilation is used. In the SPARC T3-4, however, 32-bit compilation appears to be better. In Figure 4(b), the performance scaling of the 64-bit version of our application can be seen for the Oracle SPARC T3-4. The graph shows exactly the same trend as for the 32-bit compilation (Figure 4(a)), but the absolute performance is worse than for the 32-bit compilation. The difference between our x86 based reference system and the SPARC T3-4 is that in the x64 architecture 64-bit compilation enables some additional architectural features. For the SPARCV9 architecture, however, these optimizations are already available for the 32-bit compilation. As a result, on the Oracle SPARC T3-4, 64-bit scenario-based DSE only suffers from the increased memory footprint and does not benefit from additional architectural features.

4. PROFILING THE SCENARIO-BASED DSE

To study the performance behavior in detail, we have profiled the complete scenario-based DSE framework. Figure 5(a) shows the function profile of the scenario-based DSE where 32 worker threads are running simultaneously. One of the hot spots is `ftimer`. This function only returns (after a short calculation) the current simulation time. To improve the performance, we have enabled function inlining in the Sun cc compiler for functions from external libraries (`ftimer` is a function in a dynamic shared library that is normally imported at runtime). The effect of inlining is clearly visible in the profile of Figure 5(a), where the `ftimer` function has been inlined in the 512 thread experiment. As the function `ftimer` is not present anymore in this experiment, its user time has dropped to zero. For some functions (e.g., `pearlsend` and `schedule`), however, the relative user time increases significantly. Due to the heavy use of inlined functions, the relative amount of computation time increases and, as a result, also the fraction of the total exclusive user time that is spent in these functions.

In Figure 4, we noticed that there was an increase in user and system time for situations where the number of threads was larger than 64. To analyze this phenomena, the breakdown of the total processing time accumulated over all threads is shown in Figure 5(b). Apart from the absolute values, the fractions of user, user lock and system time are more or less equal. Still, the overhead of the scenario-based DSE seems to be unreasonably large. Although none of the functions that are related to search process in scenario-based DSE show up in the profile of Figure 5(a) (all functions are related to the Sesame simulations of a single mapping), the amount of user lock time is significant. Most likely, part of this lock time is due to the lock-based job queue of the scenario-based DSE. To resolve this issue, we have redesigned our scenario-based DSE to use a lockless job queue. This design will be described in the next section.

5. THE FINAL DESIGN

The original design of the workpool of the scenario-based DSE used a queue based on mutexes and condition variables to enforce that only one thread at the time retrieves a job from the job queue. Such a lock-based design works satisfactory if the number of worker threads is low. However, when scaling up to 512 threads the contention becomes relatively large. This is substantiated by the amount of lock time in the time breakdown in Figure 5(b).

Figure 6 shows the lockless design of the workpool. The lockless implementation is largely based on volatile variables and atomic operations. Additionally, synchronization is achieved using a barrier. During execution, two stages can be distinguished: initializa-

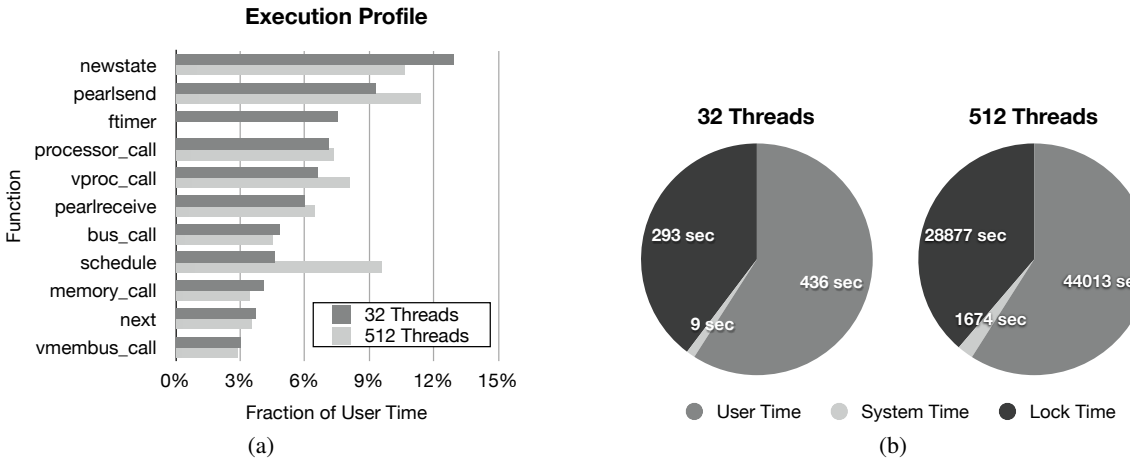


Figure 5: The (a) profile and (b) execution time breakdown (accumulated over all threads) of scenario-based DSE with 32 and 512 worker threads. In contrast to the 32 thread experiment, the 512 thread experiment uses full function inlining.

tion and the main execution. Initialization is triggered by the main thread. The main thread creates all the worker threads one by one and waits until the worker threads are initialized. After the worker threads are all ready for execution, the threads are synchronized using a barrier (BARRIER INIT).

Within the main execution, the job queue is filled and processed by the worker threads. During the filling of the queue, all the worker threads are blocked on a barrier (BARRIER START). The filling of the queue is done by the main thread. This involves the allocation of a vector with job descriptions and, next to this, the atomic variables are initialized. There are two atomic variables: 1) a pointer that refers to the first unprocessed job and 2) a pointer that refers to the last job. After initializing the queue, the main thread will also synchronize on the start barrier.

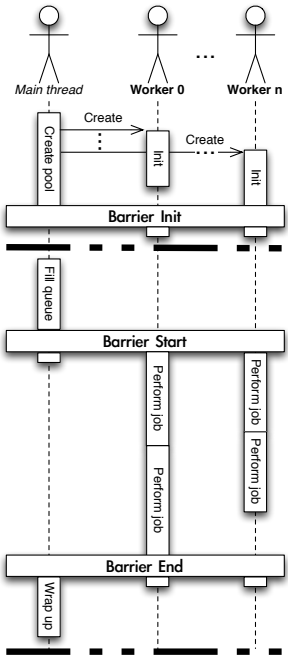


Figure 6: The "lockless" implementation of the scenario-based DSE work pool.

When all threads are started, the jobs will be processed. Each of the jobs will be handled by a single worker thread that will start a Sesame simulation in an external process. In the meanwhile, the main thread is blocked on the final barrier (BARRIER END) until the complete job queue is handled. To fetch a job, a worker thread atomically increments the pointer referring to the first unprocessed job and obtains the current value. In case the value is smaller or equal to the pointer pointing to the last job in the queue, the specific job will be fetched from the queue. Otherwise, the worker thread will also synchronize on the final barrier.

Once all the threads have reached the end barrier the main thread will wrap up. This involves the destruction of the queue and preparing it for the next batch of evaluations. In the meanwhile, the worker threads are already waiting on the start barrier. This design allows for processing multiple batches without recreating the workpool for each generation in the scenario-based DSE.

6. EXPERIMENTS

In this section, we will present the results of our final design of the scenario-based DSE using a range of different experiments. During the experiments, a fixed workload is used that consists of 1000 individual simulation jobs. The first two experiments will analyze the influence of the type of heap allocation and the type of scheduling used by the DSE framework. This is followed by an experiment that explains the increase of the user time with an increasing number of worker threads (as shown in 4). Finally, with a final experiment, we can show the scalability of the scenario-based DSE.

6.1 Heap Allocation

During the profiling of our scenario-based DSE framework, we found one hot spot with respect to system time, namely the function `take_deferred_signal()`. When further analyzing the function stack, we found that this function becomes hot due to mutex locks in `malloc` and `free`. The default malloc library on the Solaris OS that runs on the SPARC T3-4 uses a single heap for all the different threads. During system calls like `malloc` and `free`, the access to the shared heap is guarded by mutex locks. In a system where there are many hardware threads, such as the SPARC T3-4, this quickly can become a bottleneck.

Fortunately, Solaris provides more heap allocation strategies: be-

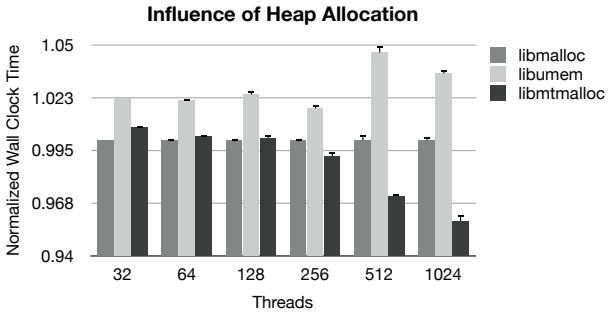


Figure 7: The influence of the different types of heap allocation on the execution time of the scenario-based DSE.

sides the default strategy, a multithreaded malloc (`libmtmalloc`) and a type of slab allocator (`libumem`). We compared the performance of our scenario-based DSE framework with these three types of heap allocation. The experiment is performed for a variable number of worker threads, where each experiment has been repeated six times. The results of the experiment are shown in Figure 7. In the results, the wall clock time of every experiment is normalized to the average wall clock time of the default heap allocation scheme and the error bars show the standard error of the mean. First, it is clear that our application is not suited for slab allocation. In the case of slab allocation, the heap allocator tries to reduce the memory fragmentation by preallocating memory slots of a certain type. When these types are allocated frequently, this quickly provides allocated memory with hardly any fragmentation. This approach may be well suited to kernel objects (In fact, `libumem` is a user space implementation of the original slab allocator inside the kernel), but in our scenario-based DSE framework `libumem` is significantly slower for all cases. It also does not solve our bottleneck problem as it has the same mutex locks as the default heap allocator.

The multithreaded heap allocator `libmtmalloc`, however, has split the heap into individual segments for each separate thread. This requires more heap space, but locally the dynamically allocated data can be created concurrently for each of the different threads without using locks. Our results show that for situations where a modest number of threads are used, `libmtmalloc` is slower. In this case, we only suffer from a larger heap space. Increasing the number of workers, `libmtmalloc` is becoming faster than the default heap allocator. The more worker threads there are, the more lock contention is present in the default heap allocator. This lock contention is not present in the multithreaded heap allocator, which is especially visible when we overload the system with 1024 worker threads.

6.2 Scheduling

Another aspect that can influence the performance is the scheduling policy of the processes. Solaris allows to set the scheduling class of a process to one of the two following classes: 1) time sharing and 2) fixed priority. Time sharing periodically recalculates the priority of a process to give each process an equal part of the processing time, whereas in the case of fixed priority it remains equal for the total lifetime of the process. As in our scenario-based DSE framework separate processes are used for each individual simulation, the scheduling can affect the performance. As shown in Figure 8(a), the desired behavior of the performance is that it improves until all the 512 hardware threads of the SPARC T3-4 are utilized. After this point, the performance should degrade very slowly. This is the case with the fixed priority policy. For time sharing, however, the performance of the scenario based DSE framework degrades faster and using 1024 threads is even slower than using 256 worker threads.

Most likely, the reason for the degraded performance of the time sharing policy is the number of context switches. In order to quantify the influence of context switches, Figure 8(b) shows the number of context switches, which indeed demonstrates a correlation between the number of context switches and the degraded performance. For the fixed priority policy, the number of context switches remains constant with an increased number of worker threads. The number of context switches for the time sharing policy, on the other hand, increases simultaneously with the number of worker threads.

It may be beneficial that each worker process keeps the affinity with the T3 core where it is running. In this way, a more efficient cache usage can be achieved. To achieve and study effect of (virtual) processor affinity, all the worker threads are bound to one of the hardware threads with the system call `processor_bind`. Here, we only performed this experiment for 512 and 1024 worker threads. In this case, it is relatively easy to spread to workers over the architecture. Looking at the performance in Figure 8(a), our simple worker mapping scheme does not give satisfactory results. Undoubtedly, better worker mapping schemes can be identified, but we do not expect to obtain significant improvements.

6.3 User Time Scaling

A much larger potential improvement can be gained if we resolve the problem related to the increase in user time. Going back to the scalability graph in Figure 4, we identified that once the number of worker threads is larger than 64, the total user time of the application starts to increase. However, as the workload remains constant, the user time should remain constant irrespective of the number of worker threads. If we can reduce the increase in user time, the total speedup of the application (compared to sequential execution) can be improved significantly.

After a thorough analysis, we have found that this problem is due to poor TLB behavior. In Figure 9, the number of TLB misses can be seen for the instruction memory and the data memory. Clearly, the number of TLB misses shows a high correlation with the increase in user time. Up to 64 worker threads, the user time was constant: the TLB experiment shows that in this case there are no TLB misses. After this point, the user time is increasing and this is reflected by a a large increase of the number of TLB misses. When a TLB miss occurs in user mode, it is also resolved in user mode. So, the time to resolve the TLB misses is also added to the user time. With more than 3.3 billion TLB misses, it is to be expected that a large increase in user time is observed.

The architecture of the SPARC T3-4 also explains why the threshold is at 64 worker threads. Each worker thread uses separate processes to perform the simulations. As a result, each worker thread needs its own private entries in the instruction and data TLBs. Up to 64 worker threads, each worker thread can run at a separate T3 core and therefore has its own TLB. However, when the number of worker threads is larger than the number of T3 cores, the TLBss are shared. In the case that all 512 hardware threads are utilized, each hardware thread can have only 8 entries in the instruction TLB and 16 entries in the data TLB.

The problem of TLB misses is hard to resolve. First, Sesame includes a large number of shared libraries. Secondly, a large amount of data is used during the simulation (e.g., input data, application and architecture models, etc.). Additionally, a large amount of output data is produced that is temporarily stored in memory.

A potential improvement is to incorporate Sesame in the evalu-

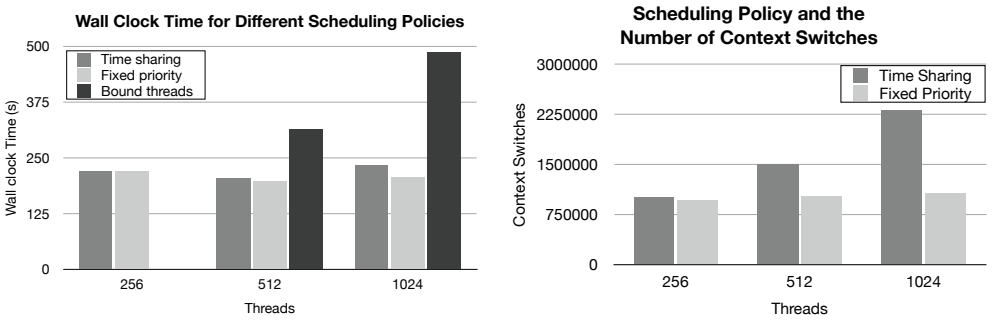
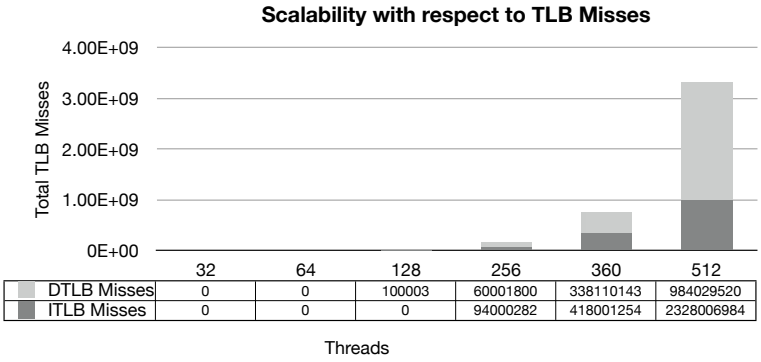


Figure 8: The (a) wall clock execution time with different scheduling policies and (b) the relation to the number of context switches.



	32	64	128	256	360	512
DTLB Misses	0	0	100003	60001800	338110143	984029520
ITLB Misses	0	0	0	94000282	418001254	2328006984

Figure 9: The number of TLB misses in relation to the number of worker threads.

ator as a shared library instead of using Sesame works in the form of separate processes. In this way, the simulation is a function call instead of an externally running process. The big advantage in this case is that shared data in simulations can be shared between all the workers. However, this would require a serious re-implementation effort (due to the large number of global variables) to turn Sesame into a loadable dynamic library.

At the moment, the only possible improvement is to increase the page size of the heap to 4MB instead of the default 8KB. This already gives a performance improvement of more than four percent with respect to the execution time with 8KB pages.

6.4 Scalability

With all the improvements described in the previous sections, we can now show the final scalability of the DSE application. For this experiment, we increased the size of the workload to $10,000$ jobs (i.e., Sesame simulations), to assure that a possible lack of sufficient workload does not limit our speedup. The results are given in Figure 10.

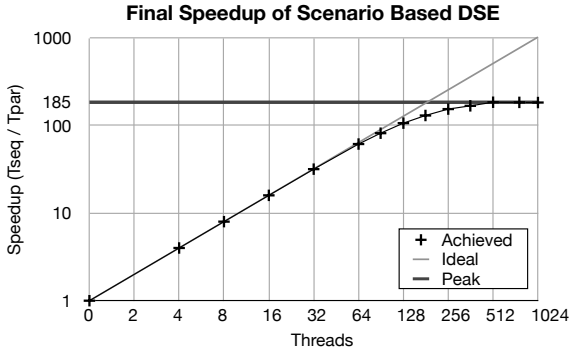


Figure 10: The final scalability of scenario-based DSE on the SPARC T3-4.

Linear speedup is achieved when the number of worker threads is less or equal to the number of T3 cores. In this case, the chip multiprocessing is exploited and most resources are private to the worker threads. Examples of these resources are the level 1 caches, the execution units and the TLBs.

For 128 threads, the parallelized evaluator is 107 times as fast as the sequential version. In this case, the average number of worker threads per T3 core is two. Each worker thread has thus its own execution unit (as there are two in each T3 core), but other resources like caches and TLBs need to be shared. As a consequence, the speedup is still close to linear.

Above the 128 threads, the execution units are also shared between the worker threads. For these configurations, the performance mostly suffers from the limited TLB size. Hence, the maximal obtained speedup is almost 185 times as fast as the sequential execution. Given the fact that there are 128 functional units in the SPARC T3-4, the chip multithreading is able to further improve the performance of our scenario-based DSE framework.

7. CONCLUSIONS

In this paper, we have studied the implementation of our scenario-based DSE framework on the SPARC T3-4. We have first profiled the execution of a non-optimized version of the scenario-based DSE framework, after which we have modified the design of the scenario based DSE in which the locks in the shared job queue have been replaced by atomic operations. The only locks left in the

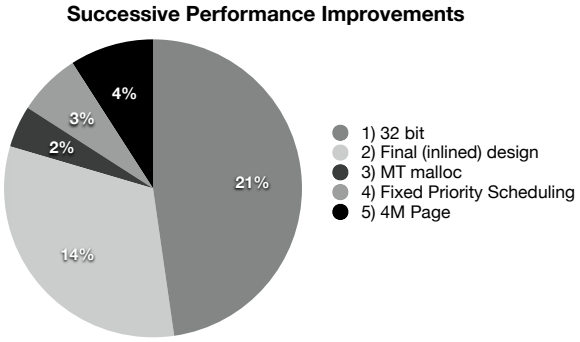


Figure 11: The successive improvements in performance after the different optimization steps. The relative improvement in execution time is given after each improvement step (starting with 32-bit compilation and ending with 4MB pages). Here, the baseline is the plain 64-bit compilation.

application are barrier synchronizations that are needed to ensure that the worker threads do not access the queue when it is being filled.

A summary of all the improvements that have performed due to the profiling of our scenario-based DSE framework on the SPARC T3-4 server is given in Figure 11. The largest improvement was made during the first two optimization steps. By simply using 32-bit instead of 64-bit compilation, a 21 percent performance improvement was already achieved. Secondly, the lockless queue implementation combined with function inlining brought another 14 percent of improvement. The heap allocation scheme and the scheduling policy, on the other hand, give moderate improvements (2 or 3%) on the final design of the scenario-based DSE framework. The largest remaining bottleneck is the high number of TLB misses. A 4MB pagesize already gives a performance improvement of more than 4%, but additional gains could be achieved in future work.

Finally, the SPARC T3-4 server gives a speedup of more than 185 times as compared to sequential execution. Given the fact that there are only 128 execution units, we can conclude that the chip multithreading approach is already paying off. Still, the SPARC T3-4 behaves poorly for a workload with a large number of (similar) processes. When all the hardware threads are utilized, only 8 instruction TLB entries are available and 16 data TLB entries. This can quickly give a performance degradation due to a large number of TLB misses.

8. REFERENCES

[1] Sparc t3-4 server. http://www.oracle.com/us/products/servers-storage/servers/sparc-enterprise/t-series/sparc-t3-4-ds-173100.pdf.

[2] Sun fire x4440 server. http://www.oracle.com/us/products/servers-storage/servers/x86/034679.pdf.

[3] C. Erbas, S. Cerav-Erbas, and A. D. Pimentel. Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design. *IEEE Transactions on Evolutionary Computation*, 10(3):358–374, June 2006.

[4] S. V. Gheorghita et al. System-scenario-based design of dynamic embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 14(1):1–45, 2009.

[5] M. Gries. Methods for evaluating and covering the design space during early design development. *Integration, the VLSI Journal*, 38(2):131–183, 2004.

[6] Z.J. Jia, A.D. Pimentel, M. Thompson, T. Bautista, and A. Nunez. Nasa: A generic infrastructure for system-level mp-soc design space exploration. In *8th IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 41–50, October 2010.

[7] G. Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[8] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998.

[9] S. Mohanty, V. K. Prasanna, S. Neema, and J. Davis. Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*, LCTES/SCOPES '02, pages 18–27, 2002.

[10] G. Palermo, C. Silvano, and V. Zaccaria. Respir: A response surface-based pareto iterative refinement for application-specific design space exploration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(12):1816–1829, 2009.

[11] A. D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Transactions on Computers*, 55(2):99–112, 2006.

[12] V. Reyes, T. Bautista, G. Marrero, P. P. Carballo, and W. Kruijtzer. Casse: A system-level modeling and design-space exploration tool for multiprocessor systems-on-chip. In *Euromicro Symposium on Digital Systems Design (DSD'04)*, pages 476–483. IEEE Computer Society, 2004.

[13] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel. Mapping on multi/many-core systems: Survey of current and emerging trends. In *Proc. of the 50th ACM/IEEE Int. Design Automation Conference (DAC '13)*, Austin, USA, June 2013.

[14] P. van Stralen and A. D. Pimentel. Scenario-based design space exploration of MPSoCs. In *Proceedings of IEEE International Conference on Computer Design (ICCD '10)*, October 2010.

[15] P. van Stralen and A. D. Pimentel. A trace-based scenario database for high-level simulation of multimedia mp-socs. In *Proc. of the Int. Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS '10)*, Samos, Greece, July 2010.

[16] P. van Stralen and A. D. Pimentel. Fast scenario-based design space exploration using feature selection. In *Proc. of the Int. Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures (PARMA'12)*, February 2012.

[17] P. van Stralen and A. D. Pimentel. Fitness prediction techniques for scenario-based design space exploration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 32(8):1240–1253, August 2013.