Resource-agnostic programming for many-core microgrids [1]

T.A.M. Bernard, C. Grelck, M.A. Hicks, C.R. Jesshope, R. Poss

University of Amsterdam, Informatics Institute, Netherlands
{t.bernard,c.grelck,m.a.hicks,c.r.jesshope,r.c.poss}@uva.nl

Abstract. Many-core architectures are a commercial reality, but programming them efficiently is still a challenge, especially if the mix is heterogeneous. Here granularity must be addressed, i.e. when to make use of concurrency resources and when not to. We have designed a data-driven, fine-grained concurrent execution model (SVP) that captures concurrency in a resource-agnostic way. Our approach separates the concern of describing a concurrent computation from its mapping and scheduling. We have implemented this model as a novel many-core architecture programmed with a language called μTC. In this paper we demonstrate how we achieve our goal of resource-agnostic programming on this target, where heterogeneity is exposed as arbitrarily sized clusters of cores.

Keywords: Concurrent execution model, many core architecture, resource-agnostic parallel programming.

1 Introduction

Many-core architectures provide the only solution to the various barriers opposing advances in mainstream computing performance [8]. However, programming applications on such platforms is still notoriously difficult [6,1,7]. Concurrency must be exposed, and in most programming paradigms it must be also explicitly managed [11]. For example, low-level constructs must be carefully assembled to map computations to hardware threads and achieve the desired synchronisation without introducing deadlocks, livelocks, race conditions, etc. From a performance perspective, any overhead associated with concurrency creation and synchronisation must be amortised with a computation of a sufficient granularity. The difficulty of the latter is under-estimated and in this paper we argue that this mapping task is too ill-defined statically and too complex to remain the programmer's responsibility. With widely varying resource characteristics, generality is normally discarded in favour of performance on a given target, requiring a full development cycle each time the concurrency granularity evolves.

[1] This work is supported by the European Union through the Apple-CORE project, grant no. FP7-ICT-215216.

We have addressed these issues in our work on *SVP* (for Self-adaptive Virtual Processor), which combines fine-grained threads with both barrier and dataflow synchronisation. Concurrency is created hierarchically and dependencies are captured explicitly. Hierarchical composition aims to capture concurrency at all granularities, without the need to explicitly *manage* it. Threads are not mapped to processing resources until run-time and the concurrency exploited depends only on the resources made available dynamically. Dependencies are captured using dataflow synchronisers and threads are only scheduled for execution when they have data to proceed. In this way, we automate thread scheduling and support asynchrony in operations. More detail on the model can be found in [3].

Asynchrony is exposed at the function level by delegating a unit of computation to independent processing resources where it can execute concurrently with its parent. It is also exposed in the dependencies captured between threads. In the context of this paper, where the model is implemented in a processor's ISA [5], we have efficient concurrency creation and synchronisation, requiring just a few processor cycles to distribute an arbitrary number of identical, indexed threads to a cluster of cores. Moreover, asynchronous operations are supported at a granularity of individual instructions and we can therefore tolerate latency in long-latency operations, such as loads from a distributed shared memory. The mapping of threads to a cluster of cores in our *Microgrid* chip architecture is automatic, and the compiled code may also express more concurrency than is available in a cluster. To resolve this mismatch, cores automatically switch from space scheduling to time scheduling when all hardware thread slots are in use. Hence, the minimal resource requirement for any SVP program is a single thread slot on a single core, which implies pure sequential execution, even though the code is expressed concurrently. It is through this technique and the latency tolerance that we achieve resource-agnostic code with predictable performance.

The main contribution of this paper is that we show simply implemented, resource agnostic SVP programs adapt automatically to the concurrency effectively available in hardware and can achieve extremely high execution efficiency. We also show that we can predict the performance of these programs based on simple throughput calculations even in the presence of non-deterministic instruction execution times. This demonstrates the effectiveness of the self-scheduling supported by SVP. In other words, we promote our research goal:

"Implement once, compile once, run anywhere."

2 The SVP concurrency model

We have built an implementation of SVP into a system language μTC and a compiler that maps this code to the Microgrid implementation. μTC is not intended as an end-user language; work is ongoing to target μTC from a data-parallel functional language (SaC [10]) and a parallelising C compiler [14,9].

In SVP programs *create* multiple threads at once as statically homogeneous, but dynamically heterogeneous *families*. The parent thread can then perform a barrier wait on termination of a named family using a *sync* action. This fork-join pattern captures concurrency hierarchically, from software component composition down to inner loops. A family is characterised by its index sequence, the initial PC for threads and the definition of unidirectional dataflow channels from, to and within the family. Channels are I-structures [2], i.e. blocking reads and single non-blocking writes; either from parent to all children ("*globals*") or sideways in the family ("*shareds*"). For more details see [5].

In the Microgrid implementation, the number of active threads per core is constrained by a block size specified for each family or by exhaustion of thread contexts. Additional expressed concurrency is then scheduled by reusing thread contexts non-preemptively. Deadlock freedom is guaranteed by restricting communication to forward-only dependency chains [17].

A key characteristic of SVP is the separation of concerns between the program and its scheduling onto computing nodes. Space scheduling is achieved by binding a collection of computing nodes, called a *place*, to a family upon its creation. This can happen at any level in the hierarchy, dynamically. Although in principle, SVP can be implemented at any level of granularity, we focus in this paper on the finest granularity, where clusters of cores implement an SVP run-time system in hardware. The SVP *create* distributes families equally to all cores in a cluster or locally depending on the place specifier. Clusters of cores are connected in rings and may be configured either at design-time or run-time.

On the Microgrid, SVP channels are mapped onto the cores' registers. Dependencies between threads mapped to the same core share the same physical registers to allow fast communication and when distributed between cores, communication is induced automatically upon register access. The latter is still a low-latency operation since constraints on dependency patterns ensure that communicating cores are adjacent on chip. Implementing I-structures on the registers also enforces scheduling dependencies between consumers and producers. Hence, long-latency operations may be allowed to complete asynchronously giving out-of-order completion with non-deterministic delay. Examples include memory operations, floating point operations (with FPU sharing between cores) and family synchronisation. This mechanism, together with support for a large number of threads per core provides the latency tolerance necessary to achieve a high utilisation of the cores' pipeline cycles. More information is available in [5].

3 An SVP implementation

The Microgrid evaluated in this paper comprises 128 cores sharing 64 FPUs with separate *add*, *mul*, *div* and *sqrt* pipelines. Each core supports up to 256 threads in 16 families using up to 1024 integer and 512 floating-point registers. On-chip memory comprises a modest 32×32KB L2 caches, shared in groups of 4 cores. There are 4 rings of 8 L2 caches; the 4 directories are connected in a top-level ring subordinated to a master directory. Two DDR3-1600 channels connect the master directory to external storage. The on-chip memory network implements a Cache-Only Memory Architecture (COMA) protocol with synchronisation at family creation, termination and on communication between threads. A cache line has no home location and migrates to the point of most recent use. This is described in more detail in [18].

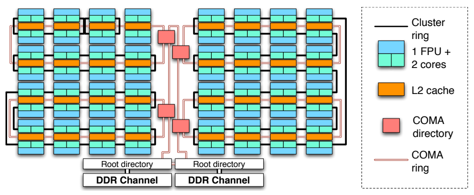


Fig. 1. Functional diagram of a 128 core Microgrid.

The following parameters are relevant to the numerical results: the two DDR channels provide 1600 million 64-bit transfers/s, i.e. a peak bandwidth of 25.6GB/s overall; each COMA ring provides a total bandwidth of 64GB/s, shared among its participants; the bus between cores and L2 caches provides 64GB/s of bandwidth; the SVP cores are clocked at 1GHz.

The Microgrid runs a minimal operating system. This includes initialisation, collection of system metrics, heap allocation, input of data from the environment through memory, and text output. A software *SVP place allocation* service allows to select dynamic cluster sizes, to subject benchmarks to heterogeneous concurrency parameters. We highlight that compiled program code is independent from all the architectural parameters of the Microgrid.

4 Experiments and results

Our aim in this paper is to show how we can obtain deterministic performance figures, even though the code is compiled from naive μTC code, with no knowledge of the target. We evaluate results from executing a range of benchmarks across a range of problem sizes on clusters of size 1-64 cores. These include both sequential and parallel algorithms with various data access patterns. The results are presented with performance on cold and warm caches. In order to analyse the performance, we need to understand the constraints on performance. For this we define two measures of arithmetic intensity (AI). The first AI_1 is the ratio of floating point operations to instructions issued. For a given kernel that is not I/O bound, this limits the floating point performance. For P cores at 1 GHz, the peak performance we can expect therefore is $P \times AI_1$. In some circumstances, we know that execution is constrained by dependencies between floating point operations and here we modify AI_1 to take this into account giving an effective intensity AI_1'. The second measure of arithmetic intensity is the ratio of Floating point operations to I/O operations, AI_2 FLOPs/Byte. I/O bandwidth IO is usually measured at the chip boundary (25.6GB/s) unless we can identify bottlenecks internally on the COMA rings (64GB/s). As these I/O bandwidths are independent of the number of cores used, this measure will provide a hard performance limit when $P \times AI_1 \geq AI_2 \times IO$.

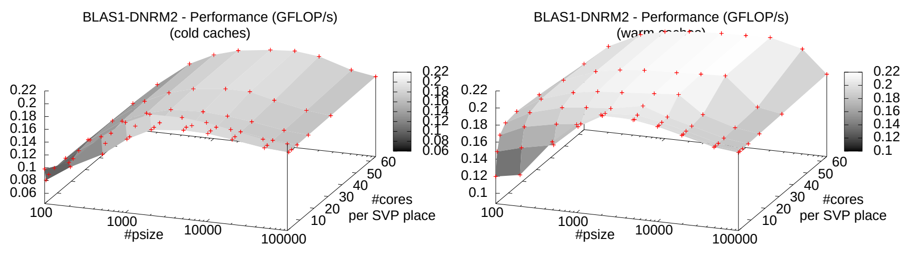
The results presented in this paper are produced using cycle-accurate emulation of a Microgrid chip that implements SVP in the ISA. It assumes custom silicon with current technology [5]. It defines all states that would exist in a silicon implementation and captures cycle-by-cycle interactions in all pipeline stages. We have used realistic multi-ported memory structures, with queueing and arbitration where we have more channels than ports. The timing assumptions are based on evaluation using CACTI [16]. We also simulate the timing of standard DDR3 channels. As details of the architecture have been described elsewhere we include only sufficient detail here to support the discussion.

4.1 Sequential code

The first kernel we consider is DNRM2 from the BLAS library, which computes the Euclidean norm of a vector. Here we do not parallelise the loop, which uses a carried dependency to calculate the sum. We are interested in how well the Microgrid tolerates the memory latency of hundreds of cycles. Branch prediction and out-of-order instruction issue can provide some latency tolerance, typically tens of cycles, which is sufficient to optimise performance when working from on-chip cache but not for larger data sets. Prefetching can do better on constant-stride accesses but as memory latencies rise, the probability that prefetched data will remain in cache diminishes. In our approach, the hardware provides latency hiding through interleaving multiple threads in the pipeline. In this kernel, a memory load and a *mul* form an independent prefix to the dependent *add* which computes the sum using a shared variable.

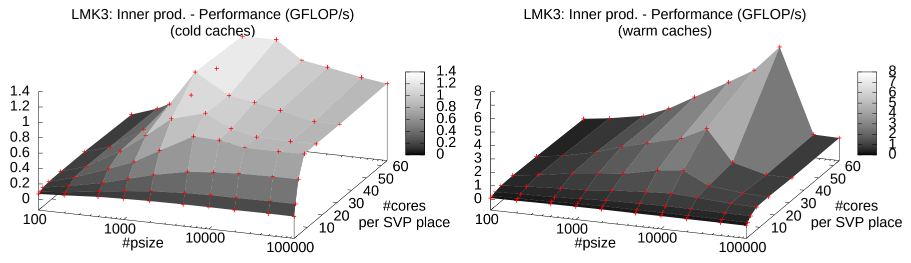
The thread code compiles to 4 instructions of which two are FP operations. So $AI_1 = 0.5$. However, every thread must wait for its predecessor to produce its result before computing its FP add. The cost of communicating the result from thread to thread requires between 6 and 11 cycles per add depending on the scheduling of threads, with the difference representing the cost of waking up a waiting thread and getting it to the read stage of the pipeline, which may be overlapped by other independent instructions in the pipeline. This implies $0.14 \leq AI'_1 \leq 0.22$, i.e. an expected single core performance of 0.14 to 0.22 GFLOP/s. As Figure 2 shows, provided we have enough threads we observe just under 0.20 GFLOP/s on one core.

We do not expect to see any performance increase by increasing the number of cores, because the independent prefix instructions that can be scheduled independently represent less than one third of the cycles required by the thread, i.e. $3 \div AI'_1$. Even with ideal scheduling and no overhead, Amdahl's law would limit speedup to a factor 1.5. The fact that we see a 10% increase is testament to the low overhead in this architecture of managing concurrency and communication.



(a) Cold caches (b) Warm caches

Fig. 2. Performance of DNRM2 on one SVP place. Working set: 8×#psize bytes.



(a) Cold caches (b) Warm caches

Fig. 3. IP performance, using N/P reduction. Working set: 16×#psize bytes.

4.2 Reductions

Any reduction can be parallelised for commutative and associative operations. The second benchmark is parallelised inner product (IP, Livermore kernel 3). The code is a straightforward extension of the naive implementation in μTC. It relies on the number of cores in the 'current place' being exposed to programs as a language primitive and splits the reduction into two stages, the first creates a family of one thread per core, which performs a local reduction and then completes the reduction between cores. When the number of threads per core is significantly larger than the number of cores, the cost of the final reduction is small and the performance should scale linearly with the number of cores. Figure 3 shows the experimental results for this code.

For IP, $AI_1 = 0.29$; however, again we must consider the effective intensity: $0.12 \leq AI'_1 \leq 0.17$, i.e. an expected single core performance of 0.12 to 0.17 GFLOP/s. The outer loop is parallel and hence we would expect a maximum performance of 0.15×64 or 9.6 GFLOP/s. However, for this code $AI_2 = 0.125$ FLOPs/byte and so performance would be memory limited to 3.2 GFLOP/s.

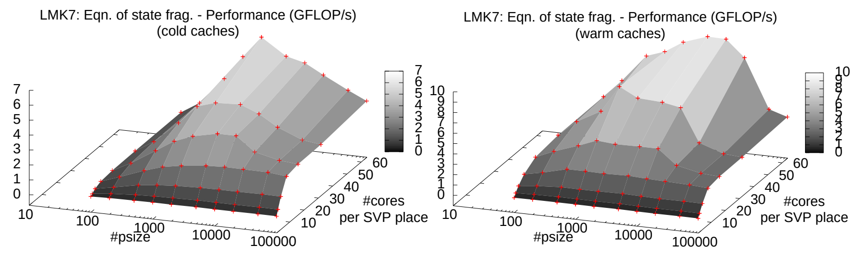
We achieve only 1.4 GFLOP/s, dropping to 0.88 GFLOP/s, for cold caches with the largest problem size. This deviation occurs when the working set does not fit in the L2 caches, because then loads to memory must be interleaved with line evictions. Even though evictions do not require I/O bandwidth, they do consume COMA ring bandwidth. It is more difficult to reason about ring bandwidth under such circumstances. In the worst case a single load may evict a cache line where the loaded line is used only by one thread before being evicted again. A single 8 byte load could require as much as two 64-byte line transfers, i.e. a perceived bandwidth for loads of 4 GB/s rising to 32GB/s if all 8 words are used. This translates into a peak performance of between 0.5 and 4 GFLOP/s with $AI_2 = 0.125$ FLOPs/Byte, when the caches become full. Note also, at a problem size of 20K on 64 cores, between 17 and 22% of the cycles required are for the sequential reduction, a large overhead and at a problem size of 100K, when this overhead is significantly smaller, only 1/6th of the problem fits in cache for up to 32 cores (1/3 for 64 cores).

With warm caches, this transition to on-chip bandwidth limited performance is delayed and more abrupt. For $P = 32$ the maximum in-cache problem size is N=16K and for $P = 64$, N=32K (ignoring code etc.). As would be expected for ring-limited performance, we see peak performance at N=10K and 20K resp. for these two cases. Any increase in problem size beyond this increases ring bandwidth to the same level as with cold caches.

4.3 Data-parallel code

We show here the behaviour of three data-parallel algorithms which exhibit different, yet typical communication patterns. Again, our μTC code is a straightforward parallelisation of the obvious sequential implementation and do not attempt any explicit mapping to hardware resources. The equation of state fragment (ESF, Livermore kernel 7) is a data parallel kernel with a high arithmetic intensity, $AI_1 = 0.48$. It has 7 local accesses to the same array data by different threads. If this locality can be exploited, then $AI_2 = 0.5$ FLOPs/Byte from off-chip memory. Matrix-matrix product (MM, Livermore kernel 21) has significant non-local access to data, in that every result is a combination of all input data. MM is based on multiple inner products and hence $AI_1 = 0.29$. However, for cache bound problems and best case for problems that exceed the cache size, $AI_2 = 3$ FLOPs/Byte from off-chip memory. Finally, FFT lies somewhere between these two extremes: it has a logarithmic number of stages that can exploit reuse but has poor locality of access. Here $AI_1 = 0.33$ and for cache-bound problems $1.6 \leq AI_2 \leq 2.9$ (logarithmic growth with problem size if there are no evictions). However, with evictions this is defined per FFT stage and $AI_2 = 0.21$.

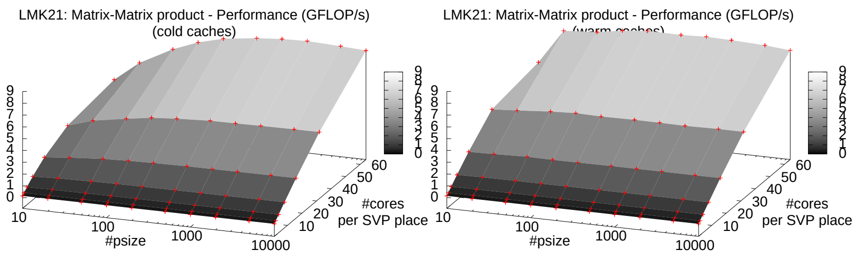
For ESF, with sufficient threads, the observed single core performance is 0.43 GFLOP/s, i.e. 90% of the expected maximum based on AI_1 for this problem (see Figure 4a). Also, while the problem is cache bound, for cold caches, we see linear speedup on up to 8 cores, 3.8 GFLOP/s. For 8 cores this problem size has 128 threads per core, reducing to 8 at 64 cores. This is an insufficient number of threads to tolerate latency and we obtain 6.6 GFLOP/s for 64 cores, 54% of the maximum limited by AI_2 (12.3 GFLOP/s). As the problem size is increased, cache evictions limit effective I/O bandwidth to 12.3GB/s at the largest problem sizes, i.e. an AI_2 constraint of around 6 GFLOP/s. We see saturation at 67% of this limit for both warm and cold caches. With warm caches and smaller



(a) Cold caches (b) Warm caches

Fig. 4. Performance of the ESF. Working set: 32×#psize bytes.

problem sizes, greater speedups can be achieved (see Figure 4b) and we achieve 9.87 GFLOP/s or 80% of the AI_2 constrained limit for a cache bound problem.



(a) Performance on cold caches (b) Performance on warm caches

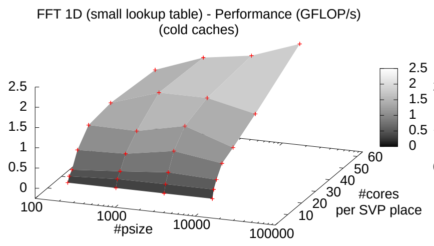
Fig. 5. Performance of the matrix-matrix product. Working set: ≈ 200×#psize bytes.

MM naively multiplies 25×25 matrices by 25×N matrices using a local IP algorithm. As $AI_2 = 3.1$ FLOPs/Byte, the I/O limit of 75 GFLOP/s exceeds the theoretical peak performance, namely 18.3 GFLOP/s. Our experiments show an actual peak of 8.57 GFLOP/s, or 47% of the maximum. As there are sufficient threads, we suspect the limit is on the COMA ring, as a significant amount of traffic is required to distribute rows and columns to cores.

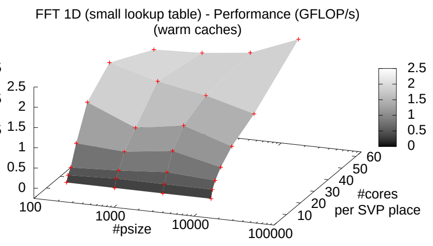
For FFT, the observed performance (cf. Figure 6) on one core is 0.23 GFLOP/s, or 78% of the AI_1 limit. When the number of cores and the problem size increase, the program becomes AI_2 constrained, as now every stage will require loads and evictions, giving an effective bandwidth of 12.3GB/s and as $AI_2 = 0.21$, an I/O constrained limit of 2.6 GFLOP/s. We observe 2.24 GFLOP/s, or 86% of this.

5 Related work

SVP addresses many-core programming from hardware thread contexts up to the programming model. In this vertical approach, it relates to XMT [13].



(a) Cold caches



(b) Warm caches

Fig. 6. Performance of the 1-D FFT. Working set: 8×#psize bytes + a lookup table.

However, the ability to define concurrency hierarchically and its data-driven scheduling bring it closer to Cilk [4] and the DDM architecture [12]. SVP differs from DDM mainly in that synchronisation is implemented in registers instead of cache, and that yet unsatisfied dependencies cause threads to suspend. Register-based synchronisation can also be found in the WaveScalar architecture [15], but WaveScalar requires pure dataflow program expression while SVP also allows thread-local sequential schedules using a regular RISC ISA.

6 Conclusion

The results presented in the previous section show efficient use of the hardware resources of single SVP places by naive implementations of computation kernels. We are able to analyse performance based on two bandwidth constrained measures and provided we have sufficient threads we observe performances that are very close (in the region of 80%) of the observed performance. Even in the worst cases we are within 50% of these predicted performances.

In conclusion, the SVP concurrency model facilitates the writing and generation of concurrent programs that need only be written and compiled once but yet can still exploit the varying parallel resources provided by particular hardware configurations. Programs can thus be expressed in the μTC language free from the restraints of resource awareness; the program only needs to express the available concurrency in algorithms and the desired synchronisations.

Acknowledgements

The development of SVP, the Microgrid architecture and the μTC compiler was initially supported by the NWO *Microgrids* project, then by the EU *Apple-CORE* project. SVP and its implementation is a group effort of the CSA group at the University of Amsterdam. The authors would like to thank especially Mike Lankamp for his work on emulation.

References

1. Amarasinghe, S.: (How) can Programmers Conquer the Multicore Menace? In: PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques. pp. 133–133. ACM, New York, NY, USA (2008)
2. Arvind, Nikhil, S., R., Pingali, K.K.: I-Structures: Data Structures for Parallel Computing. ACM Trans. Program. Lang. Syst. 11(4), 598–632 (1989)
3. Bernard, T., Bousias, K., Guang, L., Jesshope, C.R., Lankamp, M., van Tol, M.W., Zhang, L.: A General Model of Concurrency and its Implementation as Many-core Dynamic RISC Processors. In: Proc. Intl. Conf. on Embedded Computer Systems: Architecture, Modeling and Simulation, SAMOS-2008. pp. 1–9 (2008)
4. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., et al.: Cilk: an efficient multithreaded runtime system. SIGPLAN Not. 30(8), 207–216 (1995)
5. Bousias, K., Guang, L., Jesshope, C., Lankamp, M.: Implementation and Evaluation of a Microthread Architecture. J. Systems Architecture 55(3), 149–161 (2009)
6. Chapman, B.M.: The Multicore Programming Challenge. In: Advanced Parallel Processing Technologies. p. 3 (2007)
7. Gabb, H., Mattson, T., Breshears, C.: Thinking in Parallel - Three engineers' Viewpoints. Intel Software Insight Magazine 16, 24–26 (Feb 2009)
8. Geer, D.: Industry Trends: Chip Makers Turn to Multicore Processors. Computer 38(5), 11–13 (2005)
9. Grelck, C., Herhut, S., Jesshope, C., Joslin, C., Lankamp, M., Scholz, S.B., Shafarenko, A.: Compiling the Functional Data-Parallel Language SaC for Microgrids of Self-Adaptive Virtual Processors. In: 14th Workshop on Compilers for Parallel Computers (CPC'09), Zürich, Switzerland (2009)
10. Grelck, C., Scholz, S.B.: SAC: a functional array language for efficient multithreaded execution. Int. Journal of Parallel Programming 34(4), 383–427 (2006)
11. Kasim, H., March, V., Zhang, R., See, S.: Survey on Parallel Programming Model. In: Network and Parallel Computing. LNCS, vol. 5245, pp. 266–275. Springer (2008)
12. Kyriacou, C., Evripidou, P., Trancoso, P.: Data-driven multithreading using conventional microprocessors. IEEE Trans. Parallel Distrib. Syst. 17(10), 1176–1188 (2006)
13. Naishlos, D., Nuzman, J., Tseng, C.W., Vishkin, U.: Towards a first vertical prototyping of an extremely fine-grained parallel programming approach. In: SPAA '01: Proc. 13th annual ACM symposium on Parallel algorithms and architectures. pp. 93–102. ACM, New York, NY, USA (2001)
14. Saougkos, D., Evgenidou, D., Manis, G.: Specifying loop transformations for C2μTC source-to-source compiler. In: 14th Workshop on Compilers for Parallel Computers (Jan 2009)
15. Swanson, S., Schwerin, A., Mercaldi, M., Petersen, A., Putnam, A., Michelson, K., Oskin, M., Eggers, S.J.: The WaveScalar Architecture. ACM Trans. Comput. Syst. 25(2), 4 (2007)
16. Tarjan, D., Thoziyoor, S., Jouppi, N.: Cacti 4.0. Tech. rep., Western Research Laboratory, Compaq (2006)
17. Vu, T.D., Jesshope, C.R.: Formalizing SANE Virtual Processor in Thread Algebra. In: ICFEM. pp. 345–365 (2007)
18. Zhang, L., Jesshope, C.R.: On-Chip COMA Cache-Coherence Protocol for Microgrids of Microthreaded Cores. In: Bouge, et al. (eds.) Euro-Par Workshops. LNCS, vol. 4854, pp. 38–48. Springer (2007)