Mth: Codesigned Hardware/Software Support for Fine Grain Threads

David González Márquez, Adrián Cristal Kestelman, and Esteban Mocskos

Abstract—Multi-core processors are ubiquitous in all market segments from embedded to high performance computing, but only few applications can efficiently utilize them. Existing parallel frameworks aim to support thread-level parallelism in applications, but the imposed overhead prevents their usage for small problem instances. This work presents *Micro-threads* (*Mth*) a hardware-software proposal focused on a shared thread management model enabling the use of parallel resources in applications that have small chunks of parallel code or small problem inputs by a combination of software and hardware: delegation of the resource control to the application, an improved mechanism to store and fill processor's context, and an efficient synchronization system. Four sample applications are used to test our proposal: HSL filter (trivially parallel), FFT Radix2 (recursive algorithm), LU decomposition (barrier every cycle) and Dantzig algorithm (graph based, matrix manipulation). The results encourage the use of *Mth* and could smooth the use of multiple cores for applications that currently can not take advantage of the proliferation of the available parallel resources in each chip.

Index Terms—Parallel architectures, Multicore processing, Parallel programming, Multithreading

1 Introduction

The state of the art processors support 16 or 32 simultaneous threads of execution per CPU socket. The improvements in processor architectures come from optimizing its design according to the specified functionality based on different considerations such as: performance, consumption, production cost and area. Parallelism can be found at three levels: instruction-level parallelism (ILP), thread-level parallelism (TLP), and data-level parallelism (DLP).

In spite of the advances in processor computing potential, the operating systems maintain a traditional way of administering these resources. The scheduling policies do not show the same level of improvement compared with processor technology.

Moreover, only a subset of applications can make efficient use of multiple threads [2]. There are some available tools to support TLP in applications. `OpenMP` is a framework focused on scientific computing. `Cilk` is an extension to support data and task parallelism. `Pthreads` emerged as a C language threads programming interface specified by the IEEE POSIX 1003.1c standard to deal with the proprietary libraries. `Pthreads` has a simple interface that allows the programmer complete control of the threads involved in the application. The programmer has to create and synchronize threads manually, specially regarding to shared memory access. `Pthreads` gives the programmer a strong control, but the scheduling and other details are managed by the operating system runtime, which usually adds overhead and thus precludes the use of this mechanism when small parallel tasks have to be managed. The flexibility offered by `Pthreads` allows the programmer to expose more parallelism, but on the other hand it requires more programming effort and skills.

These techniques are represented on right side of the Fig. 1, which presents parallelization techniques usually employed for parallel shared memory applications. The left side of Fig. 1 shows some complex techniques to improve ILP, these techniques are limited to an instruction window size of the order of hundreds of instructions [8].

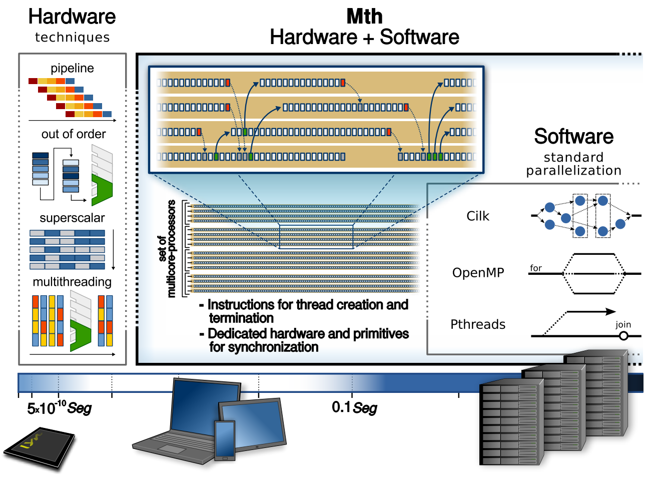


Fig. 1. The current parallelization techniques can efficiently expose parallelism at ILP level or TLP, but the gap between these two mechanisms are not treated by any existing framework or programming model. Our proposal aims to provide support to exploit parallelism at this scope.

This work presents *Micro-threads* (*Mth*) a hardware-software proposal focused on a shared thread management model. The objective is to allow the efficient execution of fine-grain parallel applications enabling the use of parallel resources in small chunks of code that could be executed in parallel. Our proposal does not require the intervention of runtime or operating system, the parallel execution is directly controlled by the application with hardware support. This proposal could smooth the use of multiple cores for applications that currently can not take advantage of the proliferation of the available parallel resources in each chip.

The speedup of a parallel program using multiple processors is limited by the time needed for the sequential fraction of the program. Moreover, the parallel implementation introduces additional overhead that constrains the achievable parallelization: if not enough work is available to be performed by a processor, the parallel implementation should take more time than the sequential. The mechanism used to spawn threads incurs additional overhead, including thread creation, deallocation and any scheduling costs [9].

Mth aims to minimize the overhead related with threads administration (creation, management and synchronization) enabling the use of parallel resources in applications that have small chunks of parallel code or small problem inputs by a combination of software and hardware: delegation of the resource control to the application, an improved mechanism to store and fill processor's context, and an efficient synchronization system.

Figure 2 shows a comparison between shows a comparison between different threading mechanisms: `Pthreads`, `OpenMP`,

- *D. González Márquez and E. Mocskos are with the Computer Science Department, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires (C1428EGA), Buenos Aires, Argentina.*
  E-mail: dmarquez@dc.uba.ar, emocskos@dc.uba.ar
- *E. Mocskos is with CSC-CONICET, Godoy Cruz 2390 (C1425FQD), Buenos Aires, Argentina.*
- *A. Cristal is with Barcelona Supercomputing Center, Instituto de Investigación en Inteligencia Artificial (IIIA-CSIC) and Department of Computer Architecture, Universitat Politècnica de Catalunya, Barcelona, España.*
  E-mail: adrian.cristal@bsc.es

Manuscript submitted: 26-Nov-2015. Manuscript accepted: 18-Aug-2016. Final manuscript received: 26-Aug-2016

`Cilk` and *Mth*. For this test, a vector is filled with random numbers obtained using a linear congruential generator. This example helps to determine the task size to be treated to overcome the overhead. The size of the array ranges from 2^6 to 2^{23} elements. Two platforms are used: i) ARM in GEM5, ii) Intel i7-920; showing the speedup related to single-core.

Figure 2 reveals that the speedup increases with the size of the problem for all the mechanisms. In the case of *Mth*, the parallel version using two threads reaches the ideal speedup with slightly more than 1000 elements, while more than 10^4 elements are needed to balance the overhead in the case of four threads. The others require more than 10^6 elements with two cores, while the size of the array should be near 10^7.

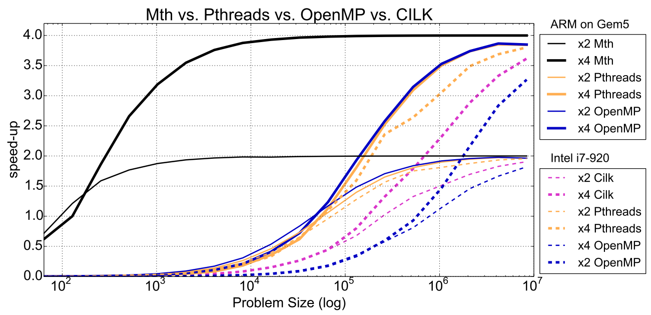


Fig. 2. Overhead of `Pthreads`, `Cilk`, `OpenMP` and *Mth*. Speed-up for 2 and 4 cores in different platforms generating random numbers.

1.1 Related Work

Price and Lowenthal [9] focus on the study of different fine-grain threading packages. In spite of analyzing outdated packages, the conclusions about the potentiality of efficient use of fine-grain parallelism are still relevant.

Zhong et al. [11] propose a new architecture for exploiting different classes of parallelism and shows that there are opportunities for exploiting fine-grain parallelism.

Madriles et al. [6] propose a design to use single-threaded applications and, through speculative threads supported by hardware, support single-threaded code. The compiler is a key component responsible for distributing instructions to cores.

Kumar et al. [5] present a full hardware scheduling solution, while Sanchez et al. [10] presents a combined hardware-software approach to build fine-grain schedulers providing direct exchange of asynchronous messages.

Fillo et al. [4] present M-Machine, a completely new processor architecture. It exposes parallelism for fixed size problems, and includes register-register communication and direct message to exploit instruction and data parallelism. Our proposal uses some of the elements presented in these works, but its main contribution is modifying minimally an existing processor (i.e. ARM) to be able to exploit parallelism even when dealing with small chunks of instructions.

2 Proposal

A process is associated with a set of threads of execution (i.e. *micro-threads*). Each micro-thread maintains information about the state of the processor from an architectural point of view (i.e. the system view offered to the programmer). The processes are managed and controlled by the operating system, but it does not have control over the associated micro-threads that are administered by each process.

2.1 System Organization

The architectural idea is proposing light modifications to existing processors to add hardware support for *Mth* while maintaining support for the standard behavior. In our proposal, the total available cores are grouped generating a two level hierarchy. Figure 3 shows the organization of the system. The first level of the hierarchy is composed by the Θ-cores, which are clusters of few cores. Inside each Θ-core, one core is selected as the *main core* (mC), which controls the rest of them (*internal cores*, iC). There are no architectural differences between the cores, distinguishing one is only to organize the system.

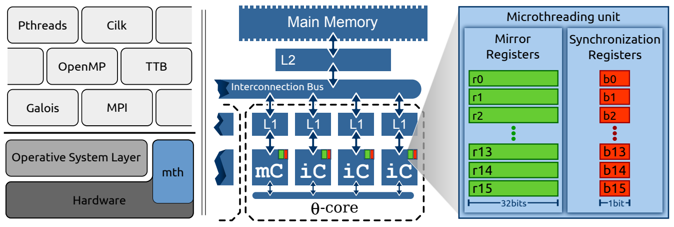


Fig. 3. *Mth* can be combined with existing frameworks to extract parallelism which can not be exploited nowadays. Each core has its own L1 cache, a current and mirror register files and a set of bits for synchronization. A Θ-core groups a set of cores connected to a memory bus and shares a L2 cache.

Each Θ-core has the same amount of internal cores (in Fig. 3 three internal cores), but this is not a requirement imposed by the design. This organization allows having different Θ-cores with different computing power and consumption. Moreover, it can be further extended to include specialized Θ-cores with hardware support for specific instructions or operations.

All cores have a L1 cache and are connected to the main memory through a L2 cache. The internal Θ-core administration is managed using a bus which connects all the cores.

The operating system dispatches each ready-to-execute process to an idle Θ-core. The application (and programmer) has the control of available resources, can manage the available threads (corresponding to the iCs), controlling what is executed in each one without the interaction of the operating system.

2.2 Architecture Extensions

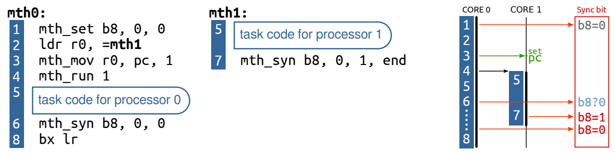
Two major extensions are included on each processor:

(i) A new mirror register file used to save the context of a process and a set of synchronization bits.
(ii) Specific instructions to control and manage the cores.

The mirror register file provides a temporary place to save the context of the next task during the setup process. It does not increment the complexity of the architecture (no need of new ports for interconnections), it only requires a simple mechanism to copy values from the mirror to the current register. It needs to contain only the information to create an execution context (program counter, stack and some parameters). Not all the registers in the system need to have a mirror register, in our implementation we use only the integer registers. The architecture provides an instruction to write values in the mirror register file in any processor. This instruction could be easily implemented, without interfering with the task currently being executed. To synchronize the tasks, 1-bit registers are included (`b0` to `b15`). In the current implementation, the new instructions work as a fence, preventing other instructions to be fetched while they are being executed, creating a bubble of 10 cycles:

- `mth_run <cpu>`: Starts a processor execution. It copies the mirror register file to the current register file. The values in the mirror register are configured during the setup and are persistent between calls to `mth_run`.

- `mth_mov <reg_src> <reg_dst> <cpu_dst>`: Moves a value stored in `reg_src` from the current processor into `reg_dst` (mirror) in the processor `cpu_dst` (different from current one). The movement can be done without interfering in the execution of `cpu_dst`.
- `mth_end`: Stops the processor execution. This instruction is executed from the current processor which will be stopped. This processor will stay in this state until a new `mth_run` is executed to start a new task.
- `mth_set <bit> <cpu_dst> <state>`: Set or clears the state of a bit in a specific core to synchronize tasks.
- `mth_syn <bit_dst> <cpu_dst> <state> <end>`: synchronizes based on the value of a 1-bit register. If `<state>` differs from the current value of `bit_dst`, it changes the value and continues. If the value is the same, the task blocks. When the optional argument `<end>` is present, the task stops after this instruction.



```
mth0:
1   mth_set b8, 0, 0
2   ldr r0, =mth1
3   mth_mov r0, pc, 1
4   mth_run 1
5   task code for processor 0
6   mth_syn b8, 0, 0
8   bx lr
```

```
mth1:
5   task code for processor 1
7   mth_syn b8, 0, 1, end
```

Fig. 4. The main task (`mth0`) executing in core 0 configures the other core to execute `mth1`. This new task is launched and a synchronization step is performed with the main task.

Figure 4 shows two tasks executing in a Θ-core using ARM and its instruction set. The main task (`mth0`) executing in core 0 configures the registers for the other task (`mth1`) in the other iC (core 1), then launches the execution of the second task and synchronizes with it:

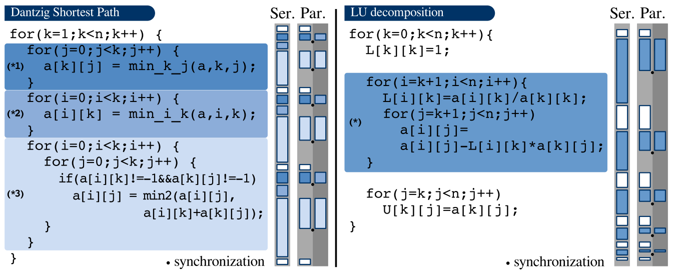
(1) Clears the bit `b8` of the synchronization register in main core.
(2) Loads the address of `mth1` in `r0`.
(3) Loads from `r0` the address of `mth1` to the PC of the mirror register of the other core.
(4) Starts the execution of `mth1`.
(5) Both tasks `mth0` and `mth1` execute.
(6) Synchronization step: `mth0` tries to clear `b8`. As this bit is in clear state, it stalls until `b8` is set.
(7) Synchronization step: `mth1` tries to set `b8` in the main core. As this bit is in clear state, it can proceed. This instruction also ends `mth1`.
(8) After synchronizing with `mth1`, this instruction can be executed.

2.3 Programming Model

From an abstract point of view, the programming model is similar to `Pthreads` with two main differences:

i) *cost of creating a new thread*: it reduces dramatically the cost of thread creation. Hardware support plays a key role in supporting this operation and decreasing the time needed to start a new thread. The thread launch consists in moving a set of registers from the mirror register file to the main one, avoiding the intervention of the operating system.
ii) *synchronization is performed actively*: the cores have the ability to wait for some clock cycles while a task is completed. This would allow decreasing the time used in waiting for some event, but the application should be programmed in such a way that a core does not have to wait for long times.

Leaving the control of the cores to the application leads to a fine tuning that can help in reducing the power consumption: having a hardware support mechanism to turn on and off a core, opens the door to use the internal cores only on demand.



```
for(k=1;k<n;k++) {
  for(j=0;j<k;j++) {
    a[k][j] = min_k_j(a,k,j);
  }
  for(i=0;i<k;i++) {
    a[i][k] = min_i_k(a,i,k);
  }
  for(i=0;i<k;i++) {
    for(j=0;j<k;j++) {
      if(a[i][k]!=-1&&a[k][j]!=-1)
        a[i][j] = min2(a[i][j],
                       a[i][k]+a[k][j]);
    }
  }
}
```

```
for(k=0;k<n;k++){
  L[k][k]=1;

  for(i=k+1;i<n;i++){
    L[i][k]=a[i][k]/a[k][k];
    for(j=k+1;j<n;j++)
      a[i][j]=
      a[i][j]-L[i][k]*a[k][j];
  }

  for(j=k;j<n;j++)
    U[k][j]=a[k][j];
}
```

Fig. 5. Examples used to test *Mth*: on the left, the computation of minimum path in a graph. On the right side, LU decomposition. The parallelization strategy and the synchronization points are shown.

3 Experiments

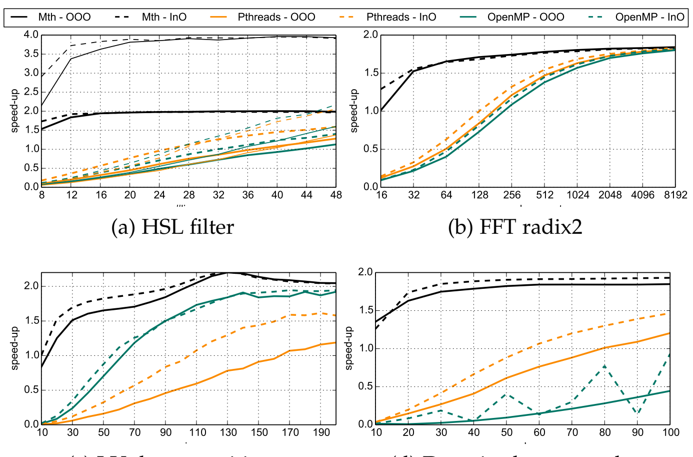
To test our proposal, GEM5 simulator [1], ARM architecture and `arm-linux-gnueabihf` compiler with maximum optimizations are used. A new *Mth* module is added to GEM5 framework. This module stores the process state information, the mirror register banks and the synchronization registers. The new instructions are implemented as synchronous operations (i.e. executes serially and flushes the pipeline). Out-of-order (`arm_detailed`, `Cortex-A9`) and in-order (`minor`, `Cortex-A8`) models are used in the examples [3]. We introduce next the applications used in this study:

- **HSL Filter**: represents a trivial parallel application that transforms from RGB to HSL color spaces by a set of successive comparison preventing the use of vectorial instructions. The parallelization strategy consists of splitting of pixels to process in all the available cores.
- **FFT Radix2**: consists in recursive independent calls over a data vector. In each call, the vector is split. After the second call, a new task is launched in parallel.
- **LU decomposition**: The right side of Fig. 5 shows the procedure [7]. Each submatrix is processed following the index k. The internal cycles compute each submatrix storing the results in a, the second one copies the results to U matrix. The parallelization strategy is based on treating both internal cycles in different cores, while the copy is performed serially.
- **Dantzig**: Shown in the left part of Fig. 5. It computes the shortest path in a graph. In each iteration, one node is added to the computation, all the paths to and from the new node are computed and the existing paths are recomputed. The parallelization consists in executing a parallel task for each of these three tasks.

The general strategy aims to deal with the code trying to mimic a guided compilation by annotated code. The algorithms have no special tuning for *Mth*, only the launch of tasks and synchronization steps are included to expose new parallelism.

4 Results and Discussion

Figure 6 compares the speedup of `OpenMP` and `Pthreads` with *Mth* while increasing the problem size for an in-order (InO) and out-of-order (OOO) processors. A remarkable fact is that the amount of data to compute in the examples is extremely small. In the case of Fig. 6a, an image with 20 lines represents 1200 bytes of data. The problem size in Fig. 6b corresponds to the vector size in double precision, while in Fig. 6c to the dimension of a double precision square matrix. The last example in Fig. 6d uses the size of the matrix obtained from a graph with the specified amount of nodes. In the case of 20 nodes, the associated matrix has 400 bytes.



(a) HSL filter

(b) FFT radix2

(c) LU decomposition

(d) Dantzig shortest path

Fig. 6. Performance evaluation of *Mth* proposal. In all the cases, highly efficient results are obtained even with small problem instances.

In all the cases, near ideal speedups are obtained even in the case of small input problem size. In the Fig. 6a, the ideal speedup is obtained for two cores in almost all the range of input data size and more than 80% of efficiency is achieved using four cores.

FFT Radix2 (Fig 6b) exceeds 75% efficiency with only 64 elements. It presents an important sequential stage, only after the second recursive call the other core is used. This prevents reaching the ideal speed-up in all the range of tested input. The input size is 2KB for 256 elements and the speed-up is 1.7X.

LU decomposition requires more input size to surpass 75% of efficiency. Cases larger than 100×100 (corresponding to 78KB of data), reaches 2X speed-up, and takes 8.56×10^8 cycles and 2.4×10^9 cycles for the out-of-order and in-order serial cases.

Dantzig presents data dependence between iterations. To start with the new iteration, the previous ones must be finished. With input size larger than 40 nodes (1.56KB of data), 1.7X and more than 80% of efficiency are obtained in out-of-order. More than 1.9X and 95% of efficiency are obtained using in-order.

```
for(istep=0;istep<S;istep++) {
  #pragma omp parallel sections
  {
    #pragma omp section
    {
      for(i=0;i<L/2;i++)
        getMd5(data+n*i);
    }
    #pragma omp section
    {
      for(i=L/2;i<L;i++)
        getMd5(data+n*i);
    }
  }
}
```

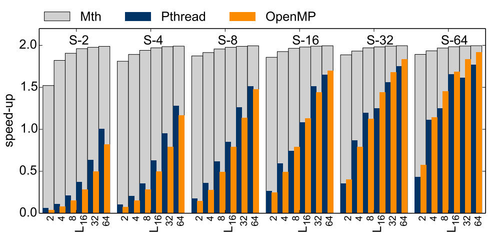


Fig. 7. Impact of initialization and synchronization using two cores: Large `S` means more work, lower `L` more synchronization.

Left part of Fig. 7 shows the `OpenMP` version of computing md5 of array elements. `L` regulates the work in each thread, while `S` increases the overall work to be done. Large `S` while keeping low `L` means that the amount of parallel work is large and a lot of synchronization is needed. The cost of initialization impacts less for large workloads (large `S`). *Mth* presents 1.8X speed-up in most of cases, the others can reach 1.8X only if `S` is large enough.

Mth is slightly affected by synchronization for `L`=2 and `S`=2 (two iterations of each cycle), but shows more than 1.8X speed-up for the rest. `OpenMP` and `Pthreads` have strong slowdown and only get a low speed-up for `S`=64.

5 Conclusions

The speedup of a parallel program using multiple processors is limited by the time needed for the sequential fraction of the program and the parallel implementation introduces additional overhead that constrains the achievable parallelization.

Micro-threads (*Mth*) aims to minimize the overhead related with threads administration (creation, management and synchronization). It includes a new mirror register file used to save the context of a process, a set of synchronization bits and specific instructions to control and manage the cores. Moreover, the programmer has complete control over the assigned group of cores which allows the efficient scheduling of chunks of parallel code.

Remarkable speedups and efficiency are obtained in all the cases, even when dealing with instance size of the order of hundreds of bytes. The obtained results encourage the use of *Mth* and show that could smooth the use of multiple cores for applications that currently can not take advantage of the multiplication of the parallel resources in each chip. Moreover, initialization and synchronization do not impact in its behavior showing remarkable speed-up in the wide range of analyzed cases.

Acknowledgments

This work was partially funded by grants from Universidad de Buenos Aires (UBACyT 20020130200096BA) and CONICET (PIP 11220110100379). The authors thank specially Osman Unsal for reading this article with fruitful criticism.

References

[1] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[2] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner, "Evolution of thread-level parallelism in desktop applications," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 302–313, Jun. 2010.

[3] F. Endo, D. Courousse, and H.-P. Charles, "Micro-architectural simulation of in-order and out-of-order arm microprocessors with gem5," in *2014 Int. Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation*, July 2014, pp. 266–273.

[4] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee, "The M-Machine multicomputer," in *Proc. of the 28th Int. Symp. on Microarch.*, Nov 1995, pp. 146–156.

[5] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: Architectural support for fine-grained parallelism on chip multiprocessors," *SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 162–173, Jun. 2007.

[6] C. Madriles, P. López, J. M. Codina, E. Gibert, F. Latorre, A. Martinez, R. Martinez, and A. Gonzalez, "Boosting single-thread performance in multi-core systems through fine-grain multi-threading," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 474–483, Jun. 2009.

[7] P. Michailidis and K. Margaritis, "Implementing parallel LU factorization with pipelining on a multicore using openmp," in *2010 IEEE 13th Int. Conf. on Comp. Sci. and Eng.*, Dec 2010, pp. 253–260.

[8] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *Proc. of the 9th HPCA*, 2003.

[9] G. W. Price and D. K. Lowenthal, "A comparative analysis of fine-grain threads packages," *J Parallel Distr Com*, vol. 63, no. 11, pp. 1050–1063, Nov. 2003.

[10] D. Sanchez, R. M. Yoo, and C. Kozyrakis, "Flexible architectural support for fine-grain scheduling," in *Proc. of the 15th ASPLOS*, 2010, pp. 311–322.

[11] H. Zhong, S. A. Lieberman, and S. A. Mahlke, "Extending multicore architectures to exploit hybrid parallelism in single-thread applications," in *Proc. of the 13th HPCA*, 2007, pp. 25–36.