Parallel Locality and Parallelization Quality

Bernard Goossens, David Parello, Katarzyna Porada, Djallal Rahmoune

Parallel Locality and Parallelization Quality

Bernard Goossens David Parello Katarzyna Porada Djallal Rahmoune

DALI, UPVD 66860 Perpignan Cedex 9 France,
LIRMM, CNRS: UMR 5506 - UM2 34095 Montpellier Cedex 5 France
firstname.lastname@univ-perp.fr

Abstract

This paper presents a new distributed computation model adapted to manycore processors. In this model, the run is spread on the available cores by *fork* machine instructions produced by the compiler, for example at function calls and loops iterations. This approach is to be opposed to the actual model of computation based on cache and predictor. Cache efficiency relies on data locality and predictor efficiency relies on the reproducibility of the control. Data locality and control reproducibility are less effective when the execution is distributed. The computation model proposed is based on a new core hardware. Its main features are described in this paper. This new core is the building block of a manycore design. The processor automatically parallelizes an execution. It keeps the computation deterministic by constructing a totally ordered trace of the machine instructions run. References are renamed, including memory, which fixes the communications and synchronizations needs. When a data is referenced, its producer is found in the trace and the reader is synchronized with the writer. This paper shows how a consumer can be located in the same core as its producer, improving *parallel locality* and *parallelization quality*. Our deterministic and fine grain distribution of a run on a manycore processor is compared with OS primitives and API based parallelization (e.g. pthread, OpenMP or MPI) and to compiler automatic parallelization of loops. The former implies (i) a high OS overhead meaning that only coarse grain parallelization is cost-effective and (ii) a non deterministic behaviour meaning that appropriate synchronization to eliminate wrong results is a challenge. The latter is unable to fully parallelize general purpose programs due to structures like functions, complex loops and branches.

1. Introduction

To run a program in parallel on a multi-core processor today, one must rewrite its C code to add by hand or through tools some OS parallelizing primitives such as *pthread*. Even with high level interfaces like OpenMP or MPI, parallelizing is not an easy job for two reasons: (i) if the resulting code is not enough synchronized, the computation is not deterministic and (ii) if it is too much synchronized, it is not parallel enough. This is illustrated by figure 1 showing the C implementation of a vector sum reduction and figure 2 showing its *pthread* implementation.

```
unsigned long sum(unsigned long array[],
                  unsigned long n){
 if (n==1) return array[0];
 if (n==2) return array[0]+array[1];
 return sum(array,n/2) + sum(&array[n/2],n-n/2);
}
```

Figure 1: A vector sum reduction: C code

The *pthread* coding is known to be tricky. A first (incorrect) version has no *pthread_join* synchronization, which highlights (i). As a result, the run is not deterministic and the returned sum may be incorrect. A second version places the first *pthread_join* synchronization on line 11 which serializes the second recursive call after the first one, which highlights (ii). As a result, the run is not parallel enough. Only the third version which places the two joins on lines 16 and 18 is satisfactory.

This example abstracts a first difficulty of hand-made parallelization with non deterministic OS primitives: achieve the exactly needed synchronization.

Edward Lee [8] writes "Threads (...) make programs absurdly nondeterministic, and rely on programming style to constrain that nondeterminism to achieve deterministic aims.". Section 2 shows how to run deterministically in parallel.

Instead of hand-parallelizing the code, the developer may rely on its compiler to automatically do loop vectorization or loop parallelization. A simple example as the program given on figure 1 cannot be automatically parallelized by *gcc*. Irregular code structures are a second problem. Section 3 explains how loops are parallelized in our approach.

A third problem of parallelization is the memory organization of the data [1]. In the *sum* example, the array to be summed is declared for example as a global variable. Hence, it is centralized when the computation is distributed. As a result, each thread brings the array pieces it needs from DRAM, where it resides. Cache can help but neighbour cores are slowed down by memory contention and the cache miss rate is impacted by the array distribution. Moreover, in a program updating shared data, keeping caches coherent requires complex hardware which slows down average memory access time. Caches and memory hierarchy, as well as branch predictors, are hardware features that rely on the principle of locality, which in essence is founded on the centralisation of data (caches) and fetched code (predictors). When the code and the data are distributed, it is the *parallel locality* which applies to data. The parallel locality principle is that a consumer should be as close as possible from its producer. In this paper we propose a measure of the producer to consumer distance, which is a way to quantify the *parallelization quality*.

```
1  typedef struct{ long *p; unsigned long  i;} ST;
2  void *sum(void *st){
3   ST str1,str2;
4   unsigned long s,s1,s2;
5   pthread_t tid1, tid2;
6   if (((ST *)st)->i>2){
7    str1.p=((ST *)st)->p;
8    str1.i=((ST *)st)->i/2;
9    pthread_create(&tid1,NULL,sum, (void *)&str1);
10   //(version 2)
11   //pthread_join(tid1,(void *)&s1);
12   str2.p=((ST *)st)->p + ((ST *)st)->i/2;
13   str2.i=((ST *)st)->i - ((ST *)st)->i/2;
14   pthread_create(&tid2,NULL,sum, (void *)&str2);
15   //(version 3)
16   //pthread_join(tid1,(void *)&s1);
17   //(versions 2 and 3)
18   //pthread_join(tid2,(void *)&s2);
19  }
20  else if (((ST *)st)->i==1){
21   s1=((ST *)st)->p[0];
22   s2=0;
23  }
24  else{
25   s1=((ST *)st)->p[0];
26   s2=((ST *)st)->p[1];
27  }
28  s=s1+s2;
29  pthread_exit((void *)s);
30 }
```

Figure 2: A vector sum reduction: *pthread* code

Section 4 defines the parallel locality and the parallelization quality and shows a programming technique to increase them. Section 5 contains a matrix multiplication program with an improved parallelization quality.

In [5], a parallelizing core hardware is proposed to distribute an execution on a manycore processor. As the parallelization is dynamic, the three problems just mentioned are more easy to tackle. The programming model presented in this paper relies on this core hardware. The next section introduces its ISA[1] extension with the *fork* instruction, its new way of fetching in parallel to build a totally ordered trace and its parallelization of the renaming process, extended to memory locations.

2. A Deterministic and Parallel Run of C Code

2.1 Deterministic Parallel Execution

The sequential execution of the code on figure 1 is deterministic. The C code fixes a total order that the run follows. The core may run machine instructions out-of-order [14] [6], i.e. in a partial order derived from Read-After-Write (RAW) register dependences, but the total order semantic is preserved.

If a parallel execution is based on a total order, it is deterministic. A totally ordered trace can be built in parallel. For example in the C *sum* function, the trace can be built top-down. Such an out-of-order construction of a totally ordered trace is possible because the control flow instructions are partially ordered. For example in the *sum* function, the control flow path in the second recursive call is independent from the control flow path in the first recursive call. This means that both calls traces can be built in parallel and orderly connected afterwards.

To avoid complications in the trace building, the hardware in [5] computes the control instructions targets rather than predicting them. Computing is slower than predicting but computing tens of branches in parallel is more efficient than predicting tens of branches in sequence, parallelism being more cost-effective than a sequential predictor, even a perfect one.

2.2 The *fork* Machine Instruction

The first 18 lines on figure 3 are an x86 translation of the *sum* C code (AT&T syntax). The *call* and *ret* instructions are replaced by *fork* and *endfork*. The *fork* instruction semantic is to keep on fetching along the continuation path, i.e. go to the *fork* instruction label. Simultaneously, a new core starts fetching along the resume path, i.e. the instruction following the *fork* in the text. The *endfork* instruction semantic is to stop fetching along the current path. Contrarily to the *call* and *ret* pair, no return address is pushed/popped.

Moreover, the given code assumes the forked path (i.e. the resume code) receives a copy of the stack pointer (i.e. x86 register rsp), meaning that both paths use the same stack area. The hardware in [5] also copies rbp, rdi, rsi and rbx. These copies are better than push/pop because a push in a function prologue and a pop in its epilogue create RAW dependences between the epilogue and the prologue of the next function call, serializing them.

In the code on figure 3, register rsi and rdi hold arguments n and $array$. The *sum* function code leaves the computed sum in register rax (lines 3, 5 and 16), from where it is read by the resume path (lines 11 and 16).

2.3 Memory Renaming

In the hardware presented in [5] the trace is run in the partial order of its dependences. False register dependences are removed by renaming [13]. Many true register dependences are also removed by copying. For example, the computation of $n - n/2$ in line 13 (figure 3) depends on registers set in lines 7 and 8. As rbx and rsi are copied to the resume path, line 13 can be run in parallel with the continuation path in line 1.

False memory dependences are also removed. For example, as the stack pointer moves back and forth (allocation in line 10 and disallocation in line 17), different code portions use the same location, creating Write-After-Read and Write-After-Write memory dependences removed by renaming.

Memory renaming schemes have been proposed [10][15], mainly to accelerate loads. The renaming relies on a predictor to quickly decide if a load depends on a previous store. However, a prediction based mechanism is not suited to eliminate false memory dependences. In [5], the memory hardware renaming is based on a search along the instruction trace total order.

Renaming is parallelized. The two recursive call destinations are simultaneously renamed. Destinations can be renamed in any order. Sources can be renamed out-of-order at the conditions that (i) the trace is totally ordered and (ii) all the destinations between a source and its producer are renamed.

The totally ordered trace is built from pieces which are fetched in parallel and later connected. When fetched, each instruction is renamed. A renamed place is allocated to hold the destination and for the sources, the closest producer is looked for by a backward search through the built trace. If a piece of the trace is missing, the search is suspended until the trace is extended. The fetch of next instructions continues during the source renaming search.

2.4 Parallel Construction of the Totally Ordered Trace

Figure 4 shows the parallel fetch of the *sum* function. The fetch is distributed on 11 cores[2]. It is done in 7 successive steps. At the first step, only core 1 is fetching. It fetches the start of the *sum* function code at line 1 (the instructions are the ones on figure 3). The rsi register holds the number n of elements to be summed, i.e. $n = 10$. Instructions on lines 1 and 2 are fetched and computed, requiring

[1] Instruction Set Architecture

[2] They may be SMT-like logical cores.

```
1  sum:  cmpq   $2, %rsi              /*if (n>2)                          */
2        ja     .L2                  /*  goto .L2                        */
3        movq   (%rdi), %rax         /*rax  = array[0]                   */
4        jne    .L1                  /*if (n==1) goto .L1                */
5        addq   8(%rdi), %rax        /*rax += array[1]                   */
6  .L1:  endfork                     /*stop                              */
7  .L2:  movq   %rsi, %rbx           /*rbx  = n                          */
8        shrq   %rsi                 /*rsi  = n/2                        */
9        fork   sum                  /*rax  = sum(array,n/2)             */
10       subq   $8, %rsp             /*allocate long on stack            */
11       movq   %rax, (%rsp)         /*save rax on top of stack          */
12       leaq   0(%rdi,%rsi,8), %rdi /*rdi  = &array[n/2]                */
13       subq   %rsi, %rbx           /*rbx  = n - n/2                    */
14       movq   %rbx, %rsi           /*n    = n - n/2                    */
15       fork   sum                  /*rax  = sum(&array[n/2],n-n/2)     */
16       addq   (%rsp), %rax         /*rax += sum(array,n/2)             */
17       addq   $8, %rsp             /*free long from top of stack       */
18       endfork                     /*stop                              */
19 init:  cmpq   $2, %rsi            /*if (n>2)                          */
20       ja     .L4                  /*  goto .L4                        */
21       movq   %rbp, (%rdi)         /*array[i]=i                        */
22       jne    .L3                  /*if (n==1) goto .L3                */
23       addq   $1, %rbp             /*i++                               */
24       movq   %rbp, 8(%rdi)        /*array[i]=i                        */
25 .L3:  endfork                     /*stop                              */
26 .L4:  movq   %rsi, %rbx           /*rbx   = n                         */
27       shrq   %rsi                 /*n     = n/2                       */
28       fork   init                 /*init(array,n/2)                   */
29       leaq   0(%rdi,%rsi,8), %rdi /*rdi   = &array[n/2]               */
30       subq   %rsi, %rbx           /*rbx  = n - n/2                    */
31       addq   %rsi, %rbp           /*rbp  = i + n                      */
32       movq   %rbx, %rsi           /*n    = n - n/2                    */
33       fork   init                 /*init(&array[n/2],n-n/2)           */
34       endfork                     /*stop                              */
35 main: movq   $0, %rbp            /*i    = 0                          */
36       movq   $10, %rsi            /*n    = 10                         */
37       movq   $array, %rdi         /*rdi  = array                      */
38       fork   init                 /*init(array,10)                    */
39       fork   sum                  /*rax = sum(array,10)               */
40       ...
```

Figure 3: A vector sum reduction (*x86* code)

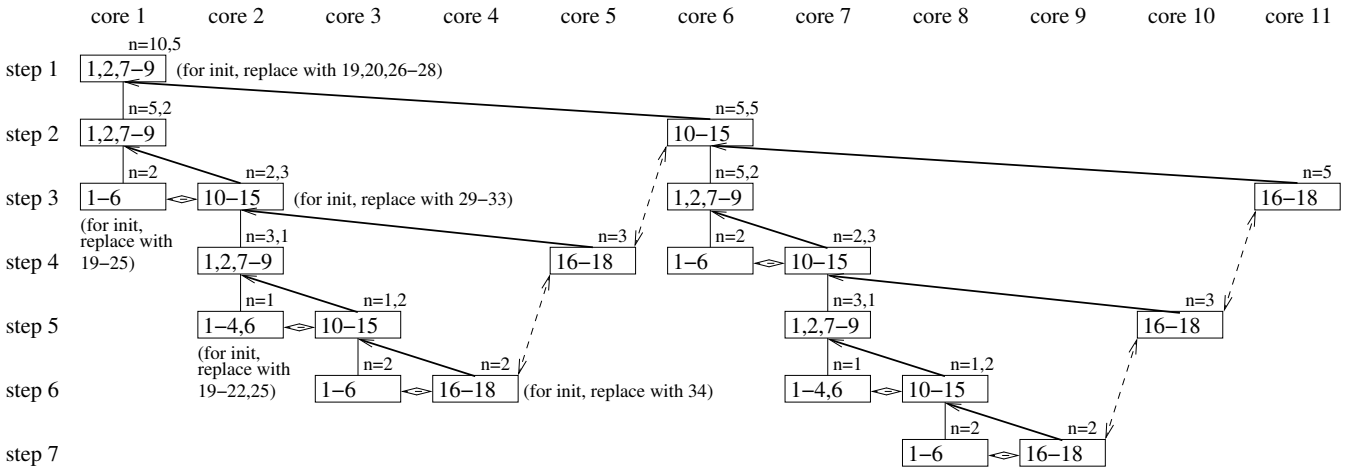

Figure 4: Trace of *sum(array, 10)*

```
#include <stdio.h>
#define SIZE 10
long array[SIZE];
void f(unsigned int i){array[i] = i;}
void for_recursive(unsigned int f,
     unsigned int l, void (*body)()){
 int n = l-f+1;
 if (n == 1){(*body)(f); return;}
 if (n == 2){(*body)(f); (*body)(f+1); return;}
 if (n != 0){
  for_recursive(f, f+n/2-1, body);
  for_recursive(f+n/2, l, body);
 }
}
void main(){
 unsigned long i; long s;
 //for (i=0; i<SIZE; i++) array[i] = i;
 //transformed into
 //for_recursive(first, last, body_function)
 for_recursive(0, SIZE-1, f);
 s=sum(array, 10);
 printf("%ld\n", s);
}
```

Figure 5: A parallelized *for* loop

no source renaming as register rsi value is known. The control is transferred to line 7. On figure 4, the upper leftmost rectangle box contains the fetched line numbers (i.e. 1, 2 and from 7 to 9). The box is labelled with the successive values of n (i.e. $n = 10$ before line 1 and $n = 5$ after line 9).

Instruction on line 9 (figure 3) is a fork. Core 6[3] (figure 4) receives the resume address from core 1 (i.e. line 10). At step 2, core 1 fetches the continuation path (line 1) and core 6 fetches the resume path (line 10). At step 3, four cores are fetching in parallel. Each core is able to compute its own control path and continue fetching independently from other cores.

The path followed by a core is a *section*. A section starts after a *fork* instruction along the resume path and ends when an *endfork* instruction is reached. On figure 4, there are 11 sections, i.e. one per core. Sections may be long (e.g. a recursive descent like the section on core 1) or short (e.g. the transmission of the partial sum on core 4). On the average, sections are around 10 instructions long resulting in a fine grain parallelization.

The hardware in [5] builds the trace total order by linking the sections. Each section is linked to its successor and predecessor (dashed bidirectional lines on figure 4)[4]. Sections belonging to the same hierarchical level are also linked (plain lines, level predecessor). For example, cores 1, 6 and 11 fetch the highest level of the *sum* function, i.e. lines 1, 2, 7 to 9 (core 1), 10 to 15 (core 6) and 16 to 18 (core 11). The three involved sections are linked (plain lines). These direct links help finding stack renamings, bypassing the subtrees. For example, instruction 16 on core 11 finds the first half sum on top of the stack, saved by instruction 11 on core 6, without waiting for the construction of the cores 7 to 10 trace sub-tree.

A lot of instructions don't even need any source renaming as their sources have known values. In the *sum* example, core 1 does not rename any source as a copy of register rsi (i.e. $rsi = 10$) is received from the *sum* function caller. Only sources referencing a production of another section need to be renamed. For example, core 6 renames rax on line 11. This renaming is to be provided by core 5 which is core 6 closest predecessor producing rax on line 16. The predecessor link from core 6 to core 5 is established at step 4, when core 5 has reached its *endfork* instruction (line 18). Core 6 sends a request to read rax to core 5 which returns the rax value to core 6 when it is computed.

In the *sum* example, there are 25 renamed sources. Ten of them concern array elements (lines 3 and 5 on figure 3). Five are references to the stack (i.e. (rsp), line 16). The last ten are references to register rax (lines 11 and 16).

The renamings of the array elements references deserve a particular treatment to avoid a long distance search of their producer in the trace. This is explained in section 4. The renamings of rax references need a round-trip communication between the renaming section and its predecessor (one way sends a request to read rax and the return way sends rax value). The renamings of the stack references $0(rsp)$ need a round-trip communication between the renaming section and its predecessor at the same hierarchical level (level predecessor). For example, line 16 in core 11 renames $0(rsp)$ from producing line 11 in core 6.

2.5 Full Renaming and Memory Management

As all destinations are renamed, the trace has a Dynamic Single Assignment form [16]. As described in [5], each core keeps its renamings in locally allocated storing resources. The cores have no data caches. The intermediate computations are kept in renaming storage until they are freed. Renaming storage is allocated up to the core capacity. When the core is full, its fetch is suspended[5].

Only the final computations are saved to physical memory, when retired. As a result, the processor memory is naturally coherent as there is a single writer.

Such parallelizing cores are simpler than actual speculative cores as they need no branch predictor, no data cache and no vector computing unit. Hence, they are better suited to be the building block of future manycore processors.

3. Parallelizing Loops

3.1 Single *for* Loops

A single *for* loop is transformed into a divide-and-conquer tree of parallel iterations. For example, figure 5 is the transformation of the array initialization loop. Figure 6 shows the x86 code produced by the compiler for the *for_recursive* function translating the *for* loop. Figure 7 shows the distributed run of the loop (only the 5 first iterations are fully shown; the 5 last ones -b5 to b9- are folded; the full tree has 9 steps distributed on 20 cores[6]).

The *body* calls can communicate (i.e. *body(j)* can import a value from *body(i)* for all $i < j$). Each *body* section has its own control flow. A parallelizable loop has independent iterations. In this case, there is no communication between the *body* sections and they all have independent control flows.

3.2 Nested Loops

Nested *for* loops are transformed into two nested divide-and-conquer trees of parallel iterations. For example, figures 8, 9 and 10 show the translation of two nested loops initializing a matrix. The x86 code is not shown but is easy to build. The run is fully parallelized with 100 sections organized as a binary tree. It is still easy for iterations of the inner loop to communicate. It is less easy for iterations of the outer loop as sections of intermediate inner loops may form a large separation of a consumer from its producer. In this case, the stack may help (the producer pushes and the consumer pops). Moreover, section 4 presents a general programming technique to optimize inter-sections communications.

[3] The core numbers are choosen arbitrarily

[4] Successor links serve to export retired data

[5] As in actual out-of-order cores when no more renaming register is free.

[6] Among which 10 do nothing more than fetching an *endfork* instruction. The compiler can eliminate such empty sections by reversing the continuation and resume paths.

```
1  for_recursive:
2        movq   %rsi, %rcx          /*rcx  = l                          */
3        subq   %rdi, %rcx          /*rcx  = l-f                        */
4        addq   $1, %rcx            /*rcx  = l-f+1                      */
5        cmpq   $1, %rcx            /*if (n!=1)                         */
6        jne    .L1                 /*  goto .L1                        */
7        fork   *%rbx               /*(*body)(f)                        */
8        endfork                    /*stop                              */
9  .L1:  cmpq   $2, %rcx            /*if (n!=2)                         */
10       jne    .L2                 /*  goto .L2                        */
11       fork   *%rbx               /*  (*body)(f)                      */
12       addq   $1, %rdi            /*rdi++                             */
13       fork   *%rbx               /*  (*body)(f+1)                    */
14       endfork                    /*stop                              */
15 .L2:  cmpq   $0, %rcx            /*if (n==0)                         */
16       je     .L3                 /*  goto .L3                        */
17       movq   %rsi, %rbp          /*rbp  = l                          */
18       shrq   %rcx                /*rcx  = n/2                        */
19       addq   %rdi, %rcx          /*rcx  = f+n/2                      */
20       subq   $1, %rcx            /*rcx  = f+n/2-1                    */
21       movq   %rcx, %rsi          /*rsi  = f+n/2-1                    */
22       fork   for_recursive       /*for_recursive(f, f+n/2-1, body)   */
23       addq   $1, %rsi            /*rsi  = f+n/2                      */
24       movq   %rsi, %rdi          /*rdi  = f+n/2                      */
25       movq   %rbp, %rsi          /*rsi  = l                          */
26       fork   for_recursive       /*for_recursive(f+n/2, l, body)     */
27 .L3:  endfork                    /*stop                              */
```

Figure 6: A *for* loop (*x86* code)

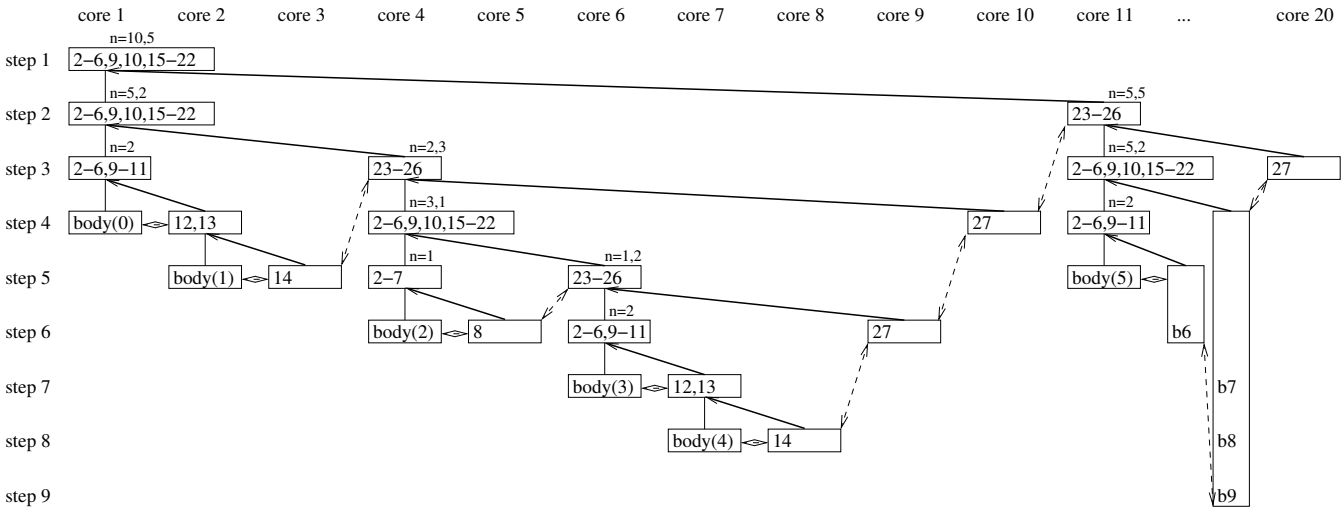

Figure 7: Trace of *for_recursive(0, 9, body)*

3.3 While loops

Figure 11 shows a *while* loop. Figure 12 shows the transformation of the *while* loop into a *for* loop. The *for* loop is then transformed in a recursive function with a continuation condition (function *cont_cond*). The run deploys a binary tree of $l - f + 1$ calls to the *for_break_recursive* function. The calls are run in parallel.

In [2], the authors give many solutions to automatically transform loops that are difficult to parallelize at compile time.

4. A Programming Technique to Increase the Quality of Parallelization

4.1 The Quality of Parallelization

A well parallelized execution should be composed of many parallel sections (fine grain is better than coarse grain), i.e. the average size s_a of the sections should be low. This allows a simultaneous fetch on all the cores, i.e. fixing the Instruction Per Cycle (IPC) peak value to the number of cores. The communications between the sections should be rare and short distance. In other words, the average number of communications per section c_a should be low and the average distance d_a from the producer to the consumer should be short, i.e. the number of visited sections should be low. Eventually, the number of instructions run n_i should be low. Among these

```
#include <stdio.h>
#define SIZE 10
int matrix[SIZE][SIZE];
void for_inner_recursive(int fo, int f, int l,
                         void (*inner_body)()){
 int n=l-f+1;
 if (n==1){(*inner_body)(fo,f); return;}
 if (n==2){
  (*inner_body)(fo,f);
  (*inner_body)(fo,f+1);
  return;
 }
 if (n!=0){
  for_inner_recursive(fo,f,f+n/2-1,inner_body);
  for_inner_recursive(fo,f+n/2,l,inner_body);
 }
}
```

Figure 8: Inner *for* loop

```
void for_outer_recursive(int fo, int lo,
     void (*outer_before)(), void (*outer_after)(),
     int fi, int li, void (*inner_body)()){
 int n=lo-fo+1;
 if (n>0){
  (*outer_before)(fo, lo);
  if (n==1) for_inner_recursive(fo,fi,li,inner_body);
  else if (n==2){
   for_inner_recursive(fo,fi,li,inner_body);
   for_inner_recursive(fo+1,fi,li,inner_body);
  }
  else{
   for_outer_recursive(fo,fo+n/2-1,
          outer_before,outer_after,fi,li,inner_body);
   for_outer_recursive(fo+n/2,lo,
          outer_before,outer_after,fi,li,inner_body);
  }
  (*outer_after)(fo, lo);
 }
}
```

Figure 9: Outer *for* loop

```
static inline void init(int i, int j){
 matrix[i][j]=i*10+j;
}
static inline void empty(int f, int l){}
main(){
 unsigned long i,j;
 /*
 for (i=0;i<10;i++)
  for (j=0;j<10;j++)
   matrix[i][j]=i*10+j;
 */
 //nested loops translated as
 //for_outer_recursive(first_outer,last_outer,
 //                    outer_before,outer_after,
 //                    first_inner,last_inner,inner_body)
 for_outer_recursive(0,9,empty,empty,0,9,init);
}
```

Figure 10: Two nested *for* loops

```
void quicksort(int f, int l){
 ...
 i1 = f; ...
 while (i1 < l && ((x1 = array[i1]) <= p)) i1++;
 ...
}
```

Figure 11: A *while* loop

```
static inline int cont_cond(int e, int p){
 return (e <= p);
}
static inline void increment(int *i){(*i)++;}
int for_break_recursive(int (*cc)(), int f, int l,
                        int p, int *x, void (*body)()){
 int n=l-f+1, i=f, i1, i2, x1, x2;
 if (!(*cc)((*x=array[f]),p)) return f;
 if (n==1){
  (*body)(&i); *x=array[i]; return i;
 }
 if (n==2){
  (*body)(&i);
  if (!(*cc)((*x=array[i]),p)) return i;
  (*body)(&i); *x=array[i]; return i;
 }
 if (n!=0){
  i1 = for_break_recursive(cc,f,f+n/2-1,p,&x1,body);
  i2 = for_break_recursive(cc,f+n/2,l,p,&x2,body);
  if (i1<f+n/2){*x=x1; return i1;}
  else{*x=x2; return i2;}
 }
}
void quicksort(int f, int l){
 ...
 //for (i1 = f;
 //     i1 < l && ((x1 = array[i1]) <= p);
 //     i1++);
 //transformed into
 i1 = for_break_recursive(cont_cond, f, l-1,
                          p, &x1, increment);
 ...
}
```

Figure 12: The *while* loop is transformed into a *for* loop with a continuation condition

four factors, d_a is the most important one as communications are expensive.

To increase the parallelization quality, we can decrease s_a (i.e. either by increasing the number of sections or by decreasing n_i). We can also decrease c_a and d_a (increasing the number of sections should not increase the average communication distance). One way to keep d_a low is to recompute a value each time it is used. This should not increase n_i too much (the recomputation should not be complex).

4.2 An Example: the *sum* Function

Figure 13 shows a *main* function calling *sum*. Figure 3 is its x86 translation. The *for* loop is translated into a parallelized function *init*. The *init* function fetched trace has the same sections decomposition as the *sum* function. Figure 4 can be adapted to the *init* function fetch. For example, the upper leftmost box corresponds to the fetch of lines 19, 20 and then from 26 to 28.

Figure 14 shows the section decomposition of the *main* function run. A starting section (upper left rectangle) is continued by the figure 4 upper leftmost rectangle (hence, lines 35 to 38 are followed by the *init* function tree on figure 4 and all belong to the first section). This section forks a new section to start the execution

```
#include <stdio.h>
#define SIZE 10
long array[SIZE];
void main(){
 unsigned long i;
 long s;
 for (i=0;i<SIZE;i++) array[i]=i;
 s=sum(array,10);
 printf("%ld\n",s);
}
```

Figure 13: A *main* function calling the *sum* function

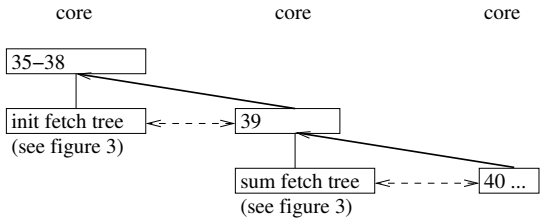

Figure 14: Trace of the *main* function

```
#include <stdio.h>
#define SIZE 10
long array[SIZE];
long sum(long array[],
         unsigned long n,
         long (*get)(),
         unsigned long i){
 if (n==1) return (*get)(array, i);
 if (n==2) return (*get)(array, i)
                + (*get)(array, i+1);
 return sum(array,n/2,get,i)
      + sum(&array[n/2],n-n/2,get,i+n/2);
}
static inline long set_array_elem(long array[],
                                  unsigned long i){
 array[i]=i;
}
static inline long get_array_elem(long array[],
                                  unsigned long i){
 set_array_elem(array,i);
 return i;
}
void main(){
 long s=sum(array,SIZE,get_array_elem,0);
 printf("s=%ld\n",s);
}
```

Figure 15: Modified *main* and *sum* functions with short producer to consumer distance

of the *sum* function (hence, the sum is computed in parallel with the array initialization). The forked section starts at line 39 and continues as the *sum* function tree on figure 4. Hence, lines 39, 1, 2, from 7 to 9, 1, 2, from 7 to 9 and from 1 to 6 belong to the same section computing array[0]+array[1].

There are 105 instructions run for the sum computation and 93 for the array initialization, hence $n_i = 198$ (neglecting the *printf* run). There are 23 sections, i.e. $s_a = 198/23 = 8.61$ (less than 9 instructions per section on the average).

There are 90 register and memory sources in the *init* run [7], all of them having a distance of 1 (i.e. requiring no communication).

There are 122 sources in the *sum* run, 15 of them having a distance of 2, 10 having a distance[8] of 12 and the remaining 97 having a distance of 1. Hence the average distance is $d_a = 337/212 = 1.59$. The average number of communications per section is $c_a = 25/23 = 1.09$ (only 25 sources among 337 need an import from another section, i.e. have a distance greater than 1).

4.3 Modifying the *sum* Function to Enhance the Quality of the Parallelization

To get the array values, function *sum* sections send renaming requests which travel along the trace, visiting one core per section in the trip, i.e. requiring 12 core-to-core communications. If these distances can be shortened, i.e. if the *sum* function section consumer can be closer to its *init* function section producer, the execution time will be reduced. Reducing d_a enhances the parallelization quality.

Figure 15 shows modified *main* and *sum* functions. A value is recomputed each time it is referenced rather than being transmitted. In manycore processors, local computation is cheaper than distant communication.

The array initialization is fused in the *sum* function. The *sum* function gets each array element. The *get* function calls a *set* function which sets the array element. Hence, each time an array element is read, it is also written. This ensures that a consumer and a producer belong to the same section.

As a general rule, a function has new arguments: a pointer on a *get* function to read variables used in the computation, a structure to encapsulate the arguments of the *get* function, a pointer on a *put* function to use computed variables and another structure to encapsulate the arguments of the *put* function.

Figure 16 shows the x86 code translation of the program depicted on figure 15.

Figure 17 shows the run trace of the modified *sum* program.

The number of instructions run is $n_i = 122$ (38% less work). There are 11 sections, i.e. $s_a = 122/11 = 11.09$. The number of sections is reduced and the average size of sections is increased (i.e. the run is artificially less parallel because the reduction of the work has reduced the number of sections). There are 146 sources, among which 15 have a distance 2 and the others have all a distance 1. The average communication distance is $d_a = 161/146 = 1.1$. The number of communications per section is $c_a = 15/11 = 1.36$.

The key factor is d_a, 31% reduced. All sections get their sources locally or from the preceding section, requiring a round-trip core-to-core communication.

Instead of having two separate computations, one to initialize the array and one to compute the sum of its elements, we have a single one, fusing the initialization into the computation. The programming technique which favours parallelization mimics a dataflow style: variables are not initialized and later used but their initialization is delayed until they are used.

When a variable is to be used multiple times, it can either be stored or recomputed. It is the compiler's job to estimate if the recomputation is better (i.e. reduces d_a enough without increasing n_i too much). To be accessed fastly, a scalar variable can be stored in the stack. To look for it, a renaming request travels along the level predecessor links, bypassing sub-trees.

[7] Including implicit sources like register $eflags$.

[8] Line 3 reads array[i] in the *sum* function from the value written by line 21 in the *init* function. The reading section is in the *sum* tree and the writing section is in the *init* tree. There are 12 sections from the writing one to the reading one.

```
1  sum:  cmpq   $2, %rsi              /*if (n>2)                          */
2        ja     .L2                  /*  goto .L2                        */
3        movq   %rsi, (%rdi)         /*inline set_array_elem             */
4        movq   (%rdi), %rax         /*inline get_array_elem             */
5        jne    .L1                  /*if (n==1) goto .L1                */
6        addq   $1, %rsi             /*inline set_array_elem             */
7        movq   %rsi, 8(%rdi)
8        addq   8(%rdi), %rax        /*inline get_array_elem             */
9  .L1:  endfork                     /*stop                              */
10 .L2:  movq   %rsi, %rbx           /*rbx  = n                          */
11       shrq   %rsi                 /*n    = n/2                        */
12       fork   sum                  /*rax  = sum(array,n/2)             */
13       subq   $8, %rsp             /*allocate long on stack            */
14       movq   %rax, (%rsp)         /*save rax on top of stack          */
15       leaq   0(%rdi,%rsi,8), %rdi /*rdi  = &array[n/2]                */
16       subq   %rsi, %rbx           /*rbx  = n - n/2                    */
17       movq   %rbx, %rsi           /*n    = n - n/2                    */
18       fork   sum                  /*rax  = sum(&array[n/2],n-n/2)     */
19       addq   (%rsp), %rax         /*rax += sum(array,n/2)             */
20       addq   $8, %rsp             /*free long from top of stack       */
21       endfork                     /*stop                              */
22 main:  movq   $0, %rbp            /*i    = 0                          */
23       movq   $10, %rsi            /*n    = 10                         */
24       movq   $array, %rdi         /*rdi  = array                      */
25       fork   sum
26       ...
```

Figure 16: The *x86* translation of the modified *sum* program

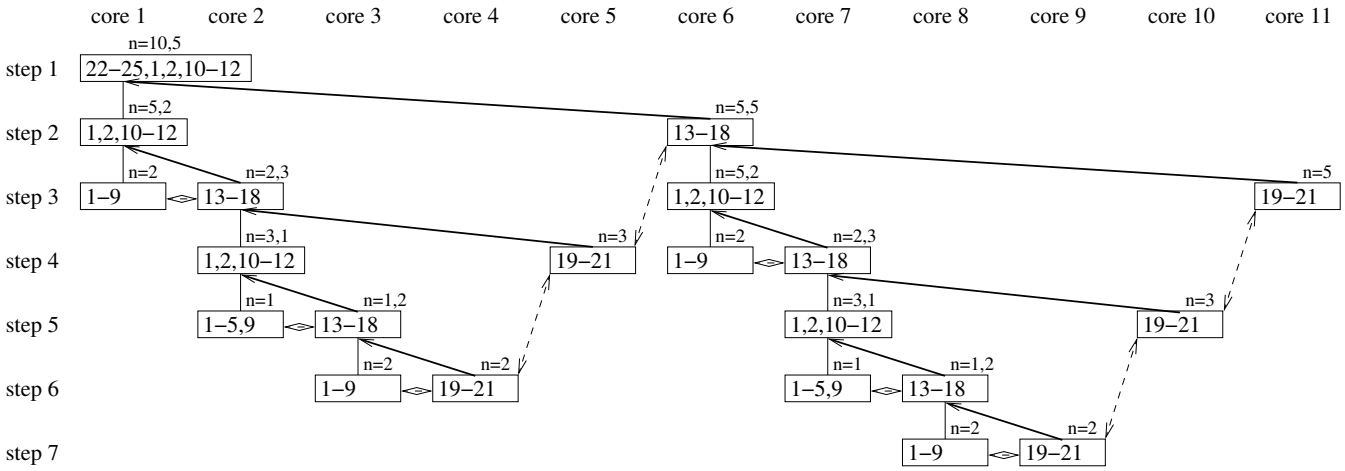

Figure 17: Trace of *sum(array, 10)* (modified *sum* program)

5. Improving a Matrix Multiplication Program

Figure 18 presents a non optimized C code to compute matrix multiplication. Optimized versions take advantage of cache locality to reduce the average access time to the input matrix elements. In a parallel environment, very distant elements are simultaneously requested (or with a short time gap) and in this case, the cache does not help. Instead, it is a better policy to keep each producer close to its consumer.

When $c[m][n] = a[m][p]*b[p][n]$ is parallelized, the run creates $m*n*p$ sections, each computing a $a[i][k]*b[k][j]$ product. Each product computation reads two elements which are looked for in the trace. Each element is set in the OS process start code which copies the values of the input matrices from the executable file to memory (it is the linker translation of lines 23 and 24 to initialize matrices a and b). To set matrix a in parallel, there are $m*p$ sections. To set matrix b in parallel, there are $p*n$ sections.

To compute $a[i][k]*b[k][j]$, we need to find $a[i][k]$ and $b[k][j]$ in the sections where they are set. The search for $b[k][j]$ visits at least $(p-k-1)*n+(n-j-1)+1 = (p-k)*n-j$ sections. To find element $a[i][k]$, it is even worse because the sections initializing b must be visited before reaching those initializing a, i.e. at least $p*n+(m-i-1)*p+(p-k-1)+1 = (n+m-i)*p-k$ sections. In the example, an element is found after an average of 10.6 visited sections. As it is by far the dominant factor in the number of imported resources, d_a is around 10 (there are $2*m*n*p$ reads to compute $m*n*p$ products).

Figures 19 and 20 are the modified implementation to reduce d_a, the distance from producer to consumer.

The first part of the code defines two types to encapsulate the needed values for the *get* and *put* functions. It also contains

```c
#include <stdio.h>
void imatmul(int *a, int *b, int *c,
 unsigned int m, unsigned int n, unsigned int p){
 //c[m][n] = a[m][p] * b[p][n]
 unsigned int i, j, k;
 for (i=0; i<m; i++)
  for (j=0; j<n; j++){
   *(c+i*n+j) = 0;
   for (k=0; k<p; k++)
    *(c+i*n+j) += *(a+i*p+k) * *(b+k*n+j);
  }
}
void
print_mat(int *a, unsigned int m, unsigned int n){
 unsigned int i,j;
 for (i=0;i<m;i++){
  for (j=0;j<n;j++)
   printf("%d ",*(a+i*n+j));
  printf("\n");
 }
}
main(){
 int a[2][3]={{1,2,3},{0,1,2}},
     b[3][4]={{2,3,4,5},{3,2,1,0},{0,1,2,3}},
     c[2][4];
 print_mat(&(a[0][0]),2,3);
 print_mat(&(b[0][0]),3,4);
 imatmul(&(a[0][0]),&(b[0][0]),&(c[0][0]),2,4,3);
 print_mat(&(c[0][0]),2,4);
}
```

Figure 18: A C program to multiply matrices

the initializing functions *set_mat_a_element* and *set_mat_b_element*. Instead of an initialization in the matrix declarations, the code provides special functions. From these, it is possible to produce in a few instructions any input matrix element in any place of the code (the *set_mat_a_elem* function fixes an element of matrix a in 4 instructions, using a table of branches issued from the *switch* control structures). The *get_mat_elem* function includes a call to the *set_mat_a_elem* or *set_mat_b_elem* function according to the referenced matrix.

The first part also contains the *sum* function, used to compute the sum part of the vector product of a line from matrix a with a column of matrix b. It is the same divide-and-conquer *sum* function as the one presented above, adapted to get the elements to be summed. The transmitted *get* function is *get_vec_elem* which reads $a[i][k]$ and $b[k][j]$ and return their products. Each section computing a product gets one element of matrix a and one element of matrix b from the same section running the *set_mat_a_elem* or *set_mat_b_elem* function. Hence, the distance d_a is no more a function of m, n and p but constant 1 for all these elements reads.

The modified example has an average distance which is ten times better than the basic one. However, each read requires 4 instructions instead of a single load in the basic trace. Hence, the number of instructions run n_i is roughly 4 times bigger.

The second part of the program is the matrix multiplication function which sets each element of the result matrix c with the vector product computed by the *sum* function. The element set is transmitted to the *put* function, i.e. the *print_mat_elem* function which prints it in a formatted manner (n elements per line).

6. Related Works, Discussion and Conclusion

Automatic parallelization parallelizes as regular as possible loops with techniques based on the polyhedral model [3].

Parallelization based on OS threads suffer from the rather high overhead of OS primitives. Using *gdb*, we have measured the cost

```c
#include <stdio.h>
typedef struct{int *a; unsigned int i;
  unsigned int j; unsigned int n;} S_MAT_ELEM;
typedef struct{int *v; int *a; unsigned int i;
  unsigned int k; unsigned int p; int *b;
  unsigned int j; unsigned int n;} S_VEC_ELEM;
int a[2][3], b[3][4], c[2][4];
static inline void
set_mat_a_elem(unsigned int i, unsigned int j){
 switch(i){
  case(0): switch(j){
    case(0): a[0][0]=1; break;
    case(1): a[0][1]=2; break;
    default: a[0][2]=3; break;
   }
  default: switch(j){
    case(0): a[1][0]=0; break;
    case(1): a[1][1]=1; break;
    default: a[1][2]=2; break;
   }
 }
}
static inline void
set_mat_b_elem(unsigned int i, unsigned int j){
 switch(i){
  case(0): switch(j){
    case(0): b[0][0]=2; break;
    case(1): b[0][1]=3; break;
    case(2): b[0][2]=4; break;
    default: b[0][3]=5; break;
   }
  case(1): switch(j){
    case(0): b[1][0]=3; break;
    case(1): b[1][1]=2; break;
    case(2): b[1][2]=1; break;
    default: b[1][3]=0; break;
   }
  default: switch(j){
    case(0): b[2][0]=0; break;
    case(1): b[2][1]=1; break;
    case(2): b[2][2]=2; break;
    default: b[2][3]=3; break;
   }
 }
}
static inline int get_mat_elem(S_MAT_ELEM *s){
 if (s->a==&a[0][0]) set_mat_a_elem(s->i,s->j);
 if (s->a==&b[0][0]) set_mat_b_elem(s->i,s->j);
 return *(s->a + s->i*s->n + s->j);
}
static inline int get_vec_elem(S_VEC_ELEM *s){
 S_MAT_ELEM st; int l,r;
 st.a=s->a; st.i=s->i; st.j=s->k; st.n=s->p;
 l=get_mat_elem(&st);
 st.a=s->b; st.i=s->k; st.j=s->j; st.n=s->n;
 r=get_mat_elem(&st);
 *(s->v)=l*r; return *(s->v);
}
int sum(int array[],unsigned int n,
        int (*get)(), S_VEC_ELEM *st,
        unsigned int i){
 if (n==1){
  st->v=&(array[0]); st->k=i;
  return (*get)(st);
 }
 if (n==2){
  int l,r;
  st->v=&(array[0]); st->k=i;
  l=(*get)(st);
  st->v=&(array[1]); st->k=i+1;
  r=(*get)(st); return l+r;
 }
 return sum(array,n/2,get,st,0)
      + sum(&array[n/2],n-n/2,get,st,n/2);
}
```

Figure 19: A C program to multiply matrices: initialization of the input matrices and vector sum

```
void imatmul(int *a, int *b, int *c, unsigned int m,
  unsigned int n, unsigned int p, void (*put)()){
 //c[m][n] = a[m][p] * b[p][n]
 unsigned int i, j; int s[p];
 S_VEC_ELEM st_v; S_MAT_ELEM st_c;
 st_v.a=a; st_v.b=b; st_v.p=p; st_v.n=n;
 st_c.a=c; st_c.n=n;
 for (i=0; i<m; i++){
  st_c.i=i; st_v.i=i;
  for (j=0; j<n; j++){
   st_c.j=j; st_v.j=j;
   *(c+i*n+j) = sum(s,p,get_vec_elem,&st_v,0);
   (*put)(&st_c);
  }
 }
}
static inline void print_mat_elem(S_MAT_ELEM *s){
 printf("%d ",*(s->a + s->i*s->n + s->j));
 if (s->j==s->n-1) printf("\n");
}
void
print_mat(int *a, unsigned int m, unsigned int n){
 unsigned int i,j; int e; S_MAT_ELEM st;
 st.a=a; st.n=n;
 for (i=0;i<m;i++){
  st.i=i;
  for (j=0;j<n;j++){
   st.j=j; e=get_mat_elem(&st); printf("%d ",e);
  }
  printf("\n");
 }
}
main(){
 print_mat(&(a[0][0]),2,3); print_mat(&(b[0][0]),3,4);
 imatmul(&(a[0][0]),&(b[0][0]),&(c[0][0]),
         2,4,3,print_mat_elem);
}
```

Figure 20: A C program to multiply matrices: matrices multiplication and *main* functions

of *pthread_create* and *pthread_join* in terms of instructions run. On an x86 processor (ubuntu 14.04), we obtained respectively 726 and 565 instructions, leading for the *sum* program example to an overhead of $1291 * \log n$ instructions for an array of n elements. The threaded version needs to sum at least 2000 terms to be cost-effective against the sequential code. On the other hand, today's hardware make it possible to send a few bytes core-to-core in a few cycles (e.g. 3 cycles -send + route + receive- for two neighbour cores on a grid with a NoC).

The existing contributions [7][9][11][12] on a hardware approach to automatize parallelization suffer from the low basic Instruction Level Parallelism (ILP). The hardware based parallelization in [5] overcomes this limitation in 3 ways: (i) very distant ILP is caught because fetch is parallelized, (ii) all false dependences are removed through full renaming and (iii) many true dependences are removed by copying values. The remaining dependences in a run are true ones related to the sequentialities of the algorithm which the program implements. In such conditions, the authors in [4] have reached a high ILP (thousands), increasing with the data size, on benchmarks issued from the PBBS suite (parallel applications; available at URL http://www.cs.cmu.edu/ pbbs/index.html).

The number of transistors on a chip allows the integration of thousands of simple cores. It is urgently needed that any program, including the OS itself, be parallelized. Parallelization should be done fastly and reliably, with reproducible computations, which is ensured if the run is deterministic.

References

[1] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 1–10, New York, NY, USA, 2008. ACM.

[2] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 283–303, Berlin, Heidelberg, 2010. Springer-Verlag.

[3] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[4] B. Goossens and D. Parello. Limits of instruction-level parallelism capture. *Procedia Computer Science*, 18(0):1664–1673, 2013. 2013 International Conference on Computational Science.

[5] B. Goossens, D. Parello, K. Porada, and D. Rahmoune. Toward a core design to distribute an execution on a manycore processor. In V. Malyshkin, editor, *Parallel Computing Technologies*, volume 9251 of *Lecture Notes in Computer Science*, pages 390–404. Springer International Publishing, 2015.

[6] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[7] H.-S. Kim and J. Smith. An instruction set and microarchitecture for instruction level distributed processing. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 71–81, 2002.

[8] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[9] M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 166–176, New York, NY, USA, 2009. ACM.

[10] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 181–193, 1997.

[11] R. Ranjan, F. Latorre, P. Marcuello, and A. Gonzalez. Fg-stp: Fine-grain single thread partitioning on multicores. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 15–24, 2011.

[12] M. Sharafeddine, K. Jothi, and H. Akkary. Disjoint out-of-order execution processor. *Transactions on Architecture and Code Optimization (TACO)*, 9(3):19:1–19:32, sept 2012.

[13] D. Sima. The design space of register renaming techniques. *IEEE Micro*, 2000.

[14] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. Dev.*, 11(1):25–33, Jan. 1967.

[15] G. S. Tyson and T. M. Austin. Memory renaming: Fast, early and accurate processing of memory communication. *Int. J. Parallel Program.*, 27(5):357–380, Oct. 1999.

[16] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, and F. Catthoor. Transformation to dynamic single assignment using a simple data flow analysis. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, pages 330–346, Berlin, Heidelberg, 2005. Springer-Verlag.