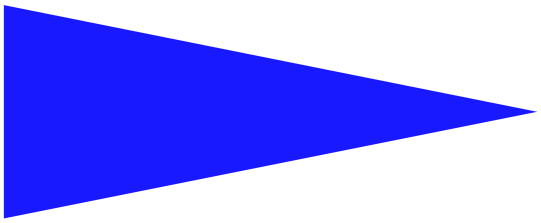
Extracting Speculative Threads Using Traces for System on a Chip

Eric Petit, François Bodin

I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1789



EXTRACTING SPECULATIVE THREADS USING TRACES FOR SYSTEM ON A CHIP

ERIC PETIT AND FRANCOIS BODIN

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

Extracting Speculative Threads Using Traces for System on a Chip

Eric Petit * and Francois Bodin **

Systèmes communicants
Projet CAPS

Publication interne n ° 1789 — Décembre 2005 — 20 pages

Abstract: This work addresses the early exploration phase, before the hardware is available, of the design of a System on a Chip. We detect threads in C programs using a software only technique. The computed threads are used as a basis for partitioning the applications. The threads are built using profiling and hot-paths information. We use a speculative model that, contrary to previous approaches, does not assume a shared memory. The speculation is performed on control flow and data structure layout. The output of the proposed method is a set of threads characterized by their execution time, the amount of memory and communication required, etc. Preliminary results show that the approach is able to capture and to characterize the main computation kernels of embedded applications.

Key-words: trace, thread, speculative, profiling, SoC, code generation, Hot-path

(Résumé : tsvp)

* eric.petit@irisa.fr
** francois.bodin@irisa.fr

Centre National de la Recherche Scientifique
(UMR 6074) Université de Rennes 1 – Insa de Rennes
Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Extraction de Threads speculatives sur la base de traces pour les SoC

Résumé : Ce travail présente la phase exploratoire de la conception d'un SoC, avant que le materiel ne soit disponible. Les threads sont detectées dans le code C avec une méthode logiciel uniquement. Les threads ainsi calculées sont utilisées comme base pour le partitionnement des applications. Les threads sont construites en utilisant des informations de profilage du code sur les hotpaths. Le modèle speculatif utilisé est, contrairement aux travaux précédent, compatible avec les systèmes à mémoire partagée. La spéculation porte sur le flot de contrôle mais aussi sur les structures de données. Le résultat de l'approche proposée est un ensemble de threads caractérisés par leur temps d'exécution, le volume de communication nécessaire, etc. Les premiers résultats montrent que l'approche est en mesure de détecter et de caractériser les principaux noyaux de calcul des applications destinées aux systèmes embarqués.

Mots clés : thread, speculative, trace, profiling, Soc, generation de code

1 Introduction

Most current embedded systems are System on a Chip that integrate multiple components with various memory and computation capabilities. The design of these SoC starts from partitioning the applications that are then mapped onto the computing units of the SoC. The partition process is very close to parallelizing the application; it extracts threads that are distributed on the computing units of the system. One difficulty that the SoC designer faces is to explore the partition possible space to find a tradeoff between performance and system cost. Speeding up the exploration requires to built new automatic tools that computes potential partitions and characterizes them qualitatively and quantitatively. To fully address SoC it is also imperative to deal with non shared memory space. Indeed, distributed memory is often chosen for SoC since shared memory implies more complex design, higher energy consumption and performance penalty.

In many cases, the input of the partition process is a sequential C program. Due to pointer aliasing issues, these programs are difficult to analyse statically. As a consequence, automatic parallelization techniques [1, 2] cannot be used. In practice, the partitioning process is often performed by "hand". This strongly limits the exploration of the design space.

In this paper, we address the issue of detecting threads in C programs that can be used as a startpoint for the partitioning of embedded applications for SoC. Figure 1 illustrates the steps involved in the processing of a sequential application. This work focuses on the exploration phase. The exploration phase is divided in four main steps. The "potential thread detection" step computes the computation intensive phases. The "thread selection and code instrumentation step" selects parts of the code that can be implemented as threads. The code is then instrumented to measure, during the execution step, the communication and precise computation load of the candidate threads. The last step, according to execution results, evaluates the characteristics of the selected threads. A feedback loop helps to refine the selection. Threads considered during this phase are speculative since they are computed based on runtime data.

The implementation phase may be performed with the help of the user. The user can use as a starting point the threads built by the exploration phase. He/she basically has to remove the "speculation".

The proposed method based on run-time data is fully automatic and, contrary to previous works [3, 4, 5, 6], consider distributed memory space. We show that the proposed process extracts "interesting" threads having low miss-speculation rate and a small ratio communication over computation.

The paper is organized as follow. In Section 2 we present the underlying assumptions on the target SoC architecture. Section 3 describes the speculative thread model used to evaluate the potential of a program partition solution. Section 4 gives an overview of the thread computation pipeline. Section 5 discusses the thread selection criteria. Section 6 reports the first experiments. In Section 7 we overview the related works. Finally, Section 8 concludes this paper.

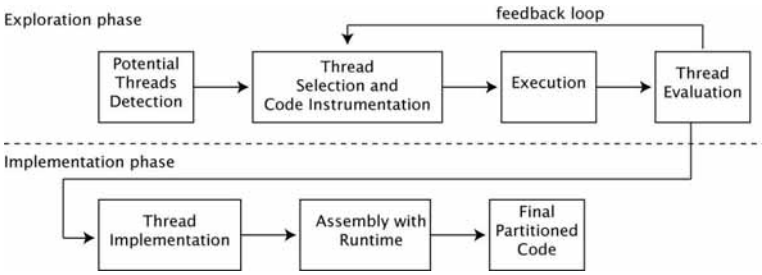


Figure 1: Partition steps.

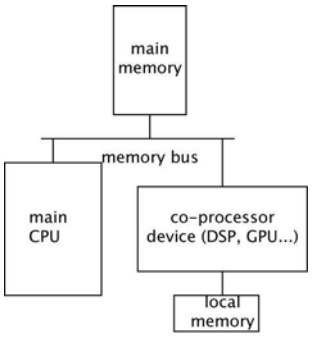


Figure 2: Target Architecture Structure.

2 Target Execution Model

The target architecture model is described in Figure 2. A main processor runs the application. A coprocessor, having it own local memory, is used to speedup the processing of some parts of the application. Typically, the main processor is a RISC micro-architecture [7, 8, 9] while the coprocessor can be a VLIW processor, an FPGA or any specialized computing unit such as a GPU [10, 9, 11].

A SoC implementing such configuration aims at exploiting each processing unit where it is the most efficient. Typically, the coprocessor runs intensive computation parts of an application.

Exploiting such a device is based on partitioning the application program in two parts and then inserting the communication and synchronization between the two programs. The partitioning must take into account the communication time as well as the type of computation to be performed on the coprocessor. For instance, if a GPU is targeted, it usually does not implement double precision floating point computation [10]. Furthermore, if the

coprocessor can be shutdown for energy saving, the time distribution of the thread activation on the coprocessor may be a criteria for selecting them.

3 Speculative Thread Model

To evaluate all the characteristics of the threads implies running efficiently many full tests on as many representative input data sets as possible. In the proposed approach, training data sets are used to compute the threads and an executable thread model is built to efficiently run the full tests. Since the threads are computed based on an execution, there is no warranty that the collected data are valid for all executions; speculation arizes at three levels:

1. At control flow level: A set of paths in the execution of the application program is assumed to be corresponding to the thread. If the execution of the thread leave the assumed set of paths, then the speculation fails.

2. At data dependency level: The data dependency between the threads and the main program must be preserved. In current study, this is not an issue since, as shown in Figure 3, we assume there is no overlap between the execution of the main program and the thread.

3. At data layout level: The data structures used in the application must be mapped in the local memory of the coprocessor. If a data access is made out of the speculated memory space the thread returns with an error code. This one of the key points of this study.

When the speculation fails the computation performed by the thread must be resumed on the main CPU. An example is shown in Figure 3. The thread is composed with basic blocks `BB1', BB2', BB3'` which are copies of `BB1, BB2, BB3` in which the data accesses have been modified according the data mapping in the local memory (see Section 3.3). In the partitioned program, the basic block `BB0` branches to a block that creates the thread, if all pre-conditions are satisfied, and copies the data in the coprocessor local memory. If the thread flow of control leaves the speculated path `BB1', BB2', BB3'`, then the thread is stopped and the main program resumes the execution at original block `BB1`. This also happens if the thread code gets out of the data space it has been allocated. If the thread terminates normally, the non local modified data are copied back to the main memory.

In the remainder of this section we detail the main features of the speculative thread model. To construct the threads we must identify memory segments accessed in the program. First we define the program working sets then we present the abstraction used to describe the data accesses. In Section 3.3 we overview the memory mapping of the data structure in the threads.

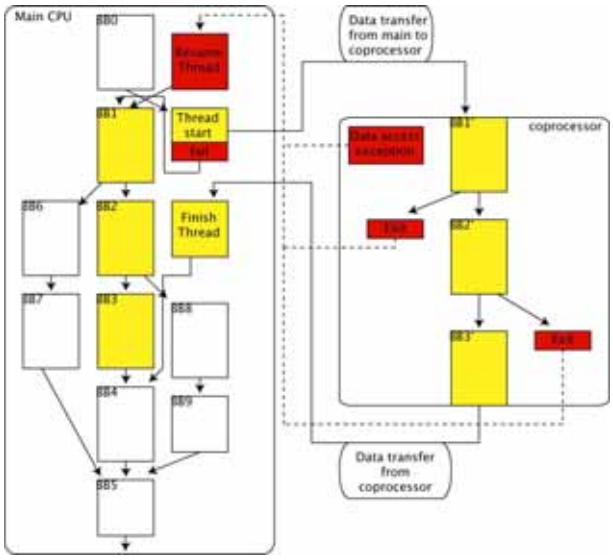


Figure 3: Thread example.

3.1 Program Working Set Description

A memory block b is defined by a memory address and a size denoted by a pair *(@,S)*. A block is created in the program whenever a new object is allocated.

To identify blocks in the program we use *abstract memory set* (AMS) that are defined by a *program creation point*, a *content type* and *a free program point* and a set of memory blocks. There is no overlap between memory blocks and a block can only belong to one AMS. This is a pre-condition to the thread launching. The set of blocks of an *abstract memory set* is updated during the program execution. The *content type* can be *value* or *address* or both.

For global variable, the *program creation point* is defined as the beginning of the main program, the *free program point* is the end of the program.

For dynamically allocated memory, the malloc statement is the *program creation point* if a unique free statement can be identified it is the *free program point* otherwise the end of the program is used.

For stack variables, the *program creation point* and *a free program point* are respectively the function entry and exit points for a given call site. The blocks corresponding to stack variables are also stacked in the abstract memory set.

For the thread construction, only set containing scalar values (i.e. no complex data structures) are considered. Examples are shown, in red, in Figure 4.

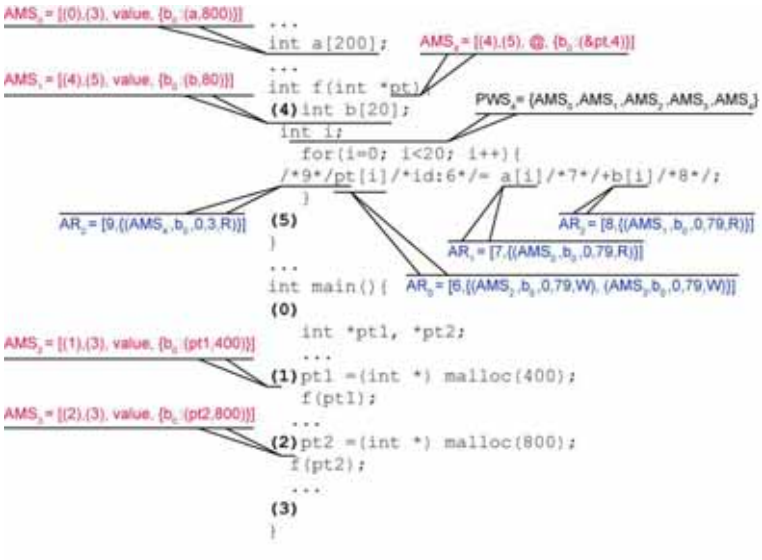


Figure 4: Abstraction example.

The set of *abstract memory sets* available at a program point during the execution defines a *program working set*. A *abstract memory set* which all memory blocks have been freed is removed from the *program working set*. An example is shown, in black, in Figure 4.

3.2 Memory Access Description

An *access* to memory, denoted *(@,R|W, s)*, is defined by the address (@), the access mode (*Read*|*Write*), and the size of the element (*s*). An *abstract reference* (AR), for a program expression, is constructed using the real accesses (obtained by instrumentation of the source code). For each access the AMS and corresponding memory block is determined. According to the block, the minimum and maximum offsets are computed. There is a unique *abstract reference* for each memory access expression in the program. The abstract reference is defined by a tuple *(id, {(abstract memory set, block, offset min, offset max, R|W) .... })* where the *id* is the identifier of the expression. Examples are shown, in blue, in Figure 4.

In the following, we assume that an abstract memory set has a unique memory block at a given time. This current restriction, as shown in Section 6, does not impact heavily for embedded applications which have limited used of dynamically allocated memory.

3.3 Thread Memory Mapping

Let assume that a set of basic blocks BB_1, ..., BB_n forms a thread. The memory elements used by the thread must be mapped onto the coprocessor unit. They are defined by the blocks corresponding to the AMS of the *abstract references* in BB_1, ..., BB_n for a given program execution. Each AMS is allocated in the coprocessor local memory. However, to avoid wasting memory and minimize memory copies, only the subset effectively accessed, given by the *(offset min, offset max)*, of the blocks is allocated. If a thread performs a memory access outside the allocated memory then there is a miss speculation and the thread is aborted. Each abstract memory set, according to the references, is added to the input and/or output sets of the thread. The mapping function is a static function that associate to an abstract memory set an address, an offset and a memory size in coprocessor local memory. The local memory going to hold the data for the thread, the offset indicates the first element to copy from the AMS to the local object. The size is the amount of memory to copy (assuming the local space is large enough). It works as follow for abstract memory sets containing values:

Global Arrays/Scalars: A memory space corresponding to a subset of the global variable is allocated onto the coprocessor memory. Only the subset of elements really accessed by the abstract references are allocated.

Local Arrays/Scalars: A corresponding variable is allocated.

Dynamically allocated variables: For dynamically allocated variables, a local array is created in the coprocessor memory.

At run time a memory mapping table is filled. The entries of the table are statically computed. However the blocks associated with an abstract memory set are updated during the execution. Figure 6 shows an example of table.

The current scheme assumes all AMS by the thread are of type *value*. Otherwise it may imply multiple indirections accesses in the threads and that would require to built a complex memory mapping. For instance, if a list is used by the thread, this would mean remapping all elements and change all pointers in the list data structures. Current scheme is not able to perform such a mapping. However, we do handle the case of scalar pointer variable which is a simple frequent case. This is detailed in the next paragraph.

3.3.1 Scalar Pointer Variable Case

Pointer scalar variables are an important special case, that can be easily handle if the following restrictions in the thread code are respected:

1. The arithmetic on the pointers is limited to adding or substracting a constant value to the pointers.
2. No multiple indirection levels.

Original code

```
    int u,m,i=0;
    int *pt1;
    int *z1= (int *) malloc(40);
    int *z2= (int *) malloc(60);
    unsigned char z3[100];
    init(z2);
    if (sscanf("%d",z2)) {
(1)    pt1=z1;
(2)    m=10;
    } else {
(3)    pt1=z2;
(4)    m=15;
    }
(5) for (;i<m;i++) {
(6)    *(pt1+4*i)= *z3;
    }
(7) u=0;
(8) free(z3);
    exit(0);
```

Data Layout

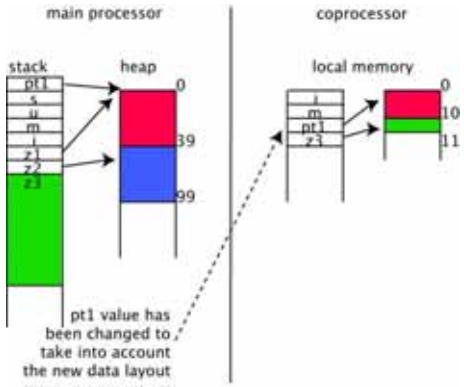


Figure 5: Statements (5) and (6) are used to build a thread. Statements (3) and (4) were not executed during the training run. On the right, the data mapping for the thread.

To implement an abstract memory set corresponding to a scalar pointer, a change in the address value is performed, so the indirect accessed in thread is translated in the thread memory space. Figure 8 shows an example of such mappings.

3.3.2 Thread Guarded Memory Access

Once the thread coprocessor memory space has been created the memory accesses must be guarded to ensure that it only addresses the abstract memory set that has been allocated onto the coprocessor. Miss speculation can happened when there is an:

Out-of-Bound access: This happens when a memory access is done outside the allocated local memory segment.

Bad access type: This happens when a read only abstract memory set is modified.

Figure 6 shows an example of guarded memory accesses. If the arrays A, B or C are not accessed within the predicted ranges defined by the abstract memory sets AMS_1, AMS_2 or AMS_3, the thread aborts its execution. This also ensure that there is no overflow of the coprocessor local memory. It should be noted here that the bound checking can easily be optimized in this case by moving the test outside the loop. The data speculation for expression $idex_2$ considers AMS_2 and AMS_3 which implies useless communications between

Memory Mapping Table

expression id	AMS	min offset	max offset
$idex_1$	$AMS_1 = (A)$	1	100
$idex_2$	$AMS_2 = (B)$	2	101
	$AMS_3 = (C)$	2	101

Original code

```
int *pt, A[...], B[...], C[...];
...
n = 1;
m = 100;
...
if (...)
    pt = B;
else
    pt = C;
...
for(i=n; i<m;i++){
    /*idex1*/A[i] = /*idex2*/pt[i+1];
}
```

Thread code

```
for(i=n; i<m;i++){
    *((type of A) check(&(A[i]),idex1)) =
       * ((type of pt) check(&(pt[i+1]),idex2));
}
```

Figure 6: Guarded memory accesses. Integer scalar variables (`i, m, n`) are omitted from the table.

the coprocessor and the main CPU. The calling context can be used to refine the speculation. This is the topic of next section.

3.4 Improving Speculation Accuracy Using History

To reduce the communication cost between the coprocessor and the main unit, it is important to distinguish the thread calling context. For such purpose we use a branch history mechanism. It is illustrated in Figure 7. In this example, the history helps distinguishing which data structures are used later in the program. To compute the history, branches are instrumented to build a vector that indicates the last executed statements. This instrumentation is used for computing the threads as well as when running the speculative threads. The history is limited in size and can be tuned for the applications. A too small history will induce more memory allocated on the coprocessor and more communication. A large history generate more execution overheads.

The abstract memory sets are sorted according to the history. The mapping function is also extended to take into account the history. In practice, this means there is one mapping table for each history value at the entry of the threads as shown in Figure 7.

Memory Mapping Table

history	expression id	AMS	min offset	max offset
H_1	$idex_1$	$AMS_1 = (A)$	1	100
	$idex_2$	$AMS_2 = (B)$	2	101
H_2	$idex_1$	$AMS_1 = (A)$	1	100
	$idex_2$	$AMS_3 = (C)$	2	101

Original code

```
int *pt, A[...], B[...], C[...];
...
n = 1;
m = 100;
...
if (...){
    H1 ;
    pt = B;
} else{
    H2 ;
    pt = C;
}
...
for(i=n; i<m;i++){
    A[i] = pt[i+1];
}
```

Thread code

```
for(i=n; i<m;i++){
    *((type of A) check(&(A[i]),idex1)) =
       * ((type of pt) check(&(pt[i+1]),idex2));
}
```

Figure 7: Use of the history to limit the data used on the coprocessor. In this case, B and C can share the same space in the coprocessor memory.

4 Extracting The Speculative Threads

In this section we overview the successive steps to construct the speculative threads. One of the key point is to obtain a process with an acceptable complexity in terms of computation time and memory space. The approach must be scalable enough to handle large applications. Figure 8 shows the overall process used. It is divided in 7 steps:

1. The first step computes the time consuming parts of the application using tools such as gprof. The complexity of this step is small.

2. The second step instruments the C source program to generate a trace of the execution. The size of the trace is limited by instrumenting only key basic blocks and using a compact trace format [12].

3. The third step, based on previous trace, computes a set of hot paths that will be the basis for choosing the statement to execute on the coprocessor. The algorithm for computing the hot paths, has been proposed in [12] and is based on suffix arrays. It

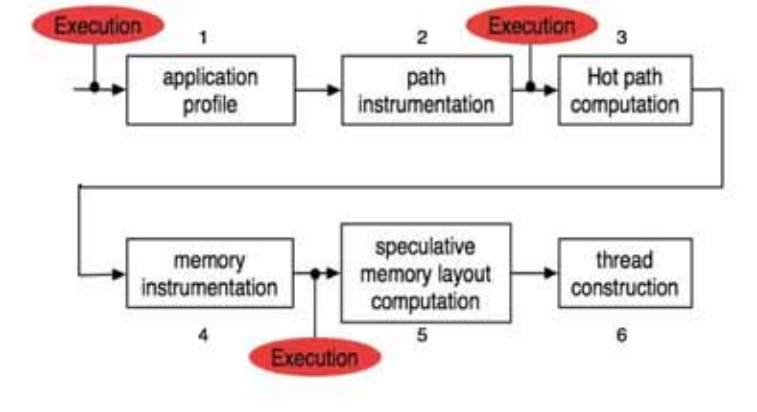


Figure 8: Thread Extraction Overview.

complexity is $O(m\ log(n))$ where m is the largest size of the thread and n the size of the trace.

4. Based on the computed hot paths, the memory accesses are instrumented to collect the abstract memory sets. For this step, the main cost is CPU time due to the instrumentation. The complexity is $O(application\ code\ statements)$ since it is proportional to the number of AMS in the code. If, due to dynamic allocation, the number of memory blocks created, for a given AMS, is too large (i.e. greater than a given constant value), the blocks are collapsed in an abstract block that is a superset of the addresses accessed.

5. From the execution of the instrumented program, the fifth step collects all abstract memory sets and associates them to the memory accesses in the thread. The complexity of this step is also linear in the size of the program.

6. Finally according to the hot paths and the speculative memory layout, the thread is constructed. The complexity of this step is linear in the size of the threads.

All the steps have been implemented using our in-house version of Sage++ [13]. In the next Section we show how this process helps application designers to choose threads for a SoC.

5 Thread Selection

When threads have been identified many parameters may be analyzed. First the *qualititative* analysis may be performed for ensuring that the threads requirements are available on the

coprocessor. The *quantitative* analysis measure on input data sets the running properties of the threads.

5.1 Qualititative Analysis

The qualitative analysis study the kind of computation performed by a thread. For instance, a coprocessor, may not provide floating point computation. This analysis may also be driven by the code generation mechanism available for the coprocessor. If the coprocessor is a GPU, a VLIW or an FPGA, the requirements will not be identical.

5.2 Quantitative Analysis

This analysis study the potential speedup the thread may provide and resources consumption for a given input data set. There is a set of overheads to take into account:

Thread initialization: It is the overhead link to start/end thread functions. In both case there is a fixed cost part due to fork/commit function.

Communication: Because we assume distributed memory system, a subset of the working set has to be copied in the coprocessor local memory. At the end, if no miss speculation arizes, there is a write back to the CPU main memory.

Miss-speculation: This overhead corresponds to restarting the original source code on the main CPU in case of miss-prediction of the thread.

Branch history computation: This overhead is due to the history instrumentation of the code.

These overheads are computed based on a target coprocessor architecture that defines the potential gain. An execution time gain is obtained if the speedup provided by a coprocessor on a given thread is such that:

$$speedup > \frac{1}{1 - \frac{VC+SC}{T_{exec}}}$$

Where VC is the variable cost, SC is the static cost and T_{exec} the thread execution time on the main processor. The VC is given by the communication time and the miss-speculation rate. The static cost corresponds to the time to start and exit the thread and the history computation.

The criteria related to the resources consumption are the following:

Code size: The code size is not directly available since it depends on the chosen coprocessor. Because we explicitly construct the code of the thread, it is possible to get an estimation of the amount of instruction memory space using a cross compiler.

Data memory size: The thread extraction pipeline computes the amount of data memory needed on the coprocessor to execute the thread.

Thread activation pattern: A thread is also characterized by its activation pattern. Because we are using traces, we can compute the delays between the activations of the thread. The delays, if large enough, may be used to turnoff the coprocessor to save energy or to allocate its computing power to another application.

6 Preliminary Experiments

In this section we present preliminary experiments. These experiments aims at checking that the proposed process can extract "interesting" threads having low miss-speculation rate and a small ratio communication versus computation. Furthermore, we want to check that the computed threads are robust to data input changes.

We use 12 small benchmarks from UTDSP [14] and Powerstone benchmarksuite [15]. Those programs are mainly composed of small multimedia applications that address image encoding, mpeg decoding, signal processing, etc.

Extracting the threads on a PC workstation was performed in the range of 1s to 10 minutes. To evaluate the thread characteristics we proceed as described in Figure 9. We use one execution to evaluate the miss-speculation rate. For this execution, the thread code is instrumented for checking control flow and data accesses as described in Section 3. A second execution is used to estimate the thread CPU time. For this, only the entry and exit points are lightly instrumented based on the processor cycle counter. Performing in two passes avoids time distortion due to instrumentation.

Figure 10 reports the thread coverage of the applications. This coverage is obtained using one up to 5 threads. In average 59,7 % of execution time are cover by threads. The average communication volume at the activation of the thread is 500 bytes. For all the applications the miss-speculation rate is small enough to be ignored.

As it can be seen in Figure 10, the applications exhibit three classes of behavior. The first class is composed by application where the threads cover more than 80% of the total execution time. The second one has a thread coverage in the range of 50% to 70% and the last class contains applications with a coverage inferior to 30%.

Tables 1, 2 and 3 reports detailed data for a representative example of each class. The reported communication time estimation is computed on a 3,00 GHz Pentium 4 architecture. The fixed cost of the considered `memcpy` C function call is 200 cycles, and the variable cost per byte of 0,6 cycle:

$$T_{comm} = \frac{2 * \#Activations * (Comm.\ Volume * 0,6 + 200)}{Processor\ Freq\ in\ Hz}$$

The factor 2 is due to the data copy at entry and exit point.

Table 1 gives the results of experiments for jpeg from Powerstones. The total execution time coverage for jpeg threads is about 85% for 17,5% of the application C code. Thread

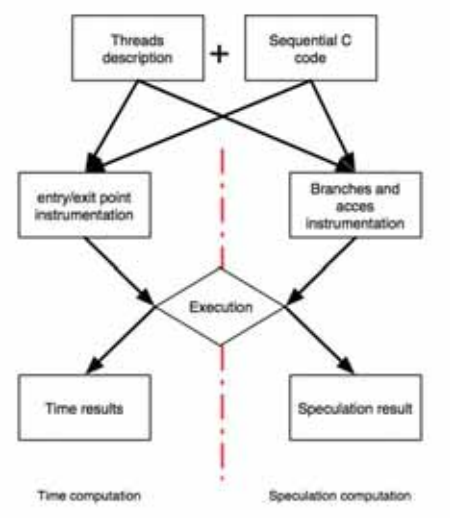


Figure 9: Experimentation process

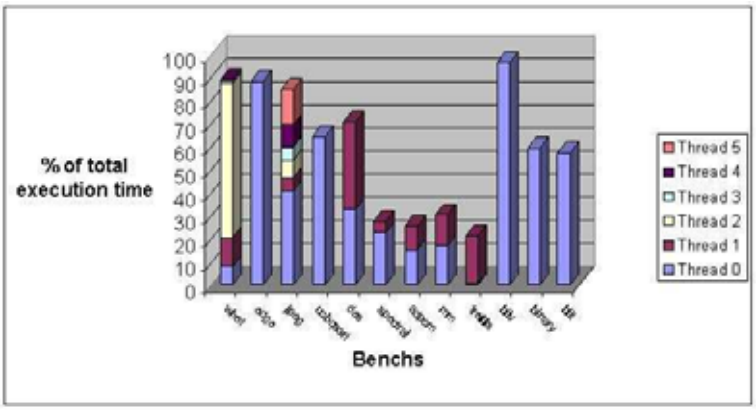


Figure 10: Threads total coverage of execution time for all tested benches

5 exhibits a large communication/execution time ratio indicating a poor potential benefit. Thread 1 is beneficial if the speedup provided by the coprocessor on the thread code is greater than 1.47. The other threads have an interesting very small communication/execution time

jpeg	# Activations	Execution Time Coverage (%)	Code Coverage (%)	Comm. Volume	$\frac{T_{comm}}{T_{exec}}$	Average Activation Distance (nanosec.)
Thread 0	600	40,97	12	504	0,02	42
Thread 1	4800	5,66	0.5	128	0,30	43
Thread 2	600	7,66	0,5	383	0,09	1300
Thread 3	600	5,77	1	280	0,14	1300
Thread 4	3180	9,46	2	17	0,12	438
Thread 5	19228	15,19	1,5	17	1,02	86

Table 1: Threads statistics for a highly covered application.

DES	# Activations	Execution Time Coverage (%)	Code Coverage (%)	Comm. Volume (bytes)	$\frac{T_{comm}}{T_{exec}}$	Average Activation Distance (nanosec.)
Thread 0	5577	32,68	8	159	0,16	2151
Thread 1	5577	38,14	9	176	0,19	2355

Table 2: Threads statistics for an average coverage of the application.

adpcm	# Activations	Execution Time Coverage (%)	Code Coverage (%)	Comm. Volume (bytes)	$\frac{T_{comm}}{T_{exec}}$	Average Activation Distance (nanosec.)
Thread 0	256	15,39	2,5	49	0,35	3797
Thread 1	256	10,17	1,5	33	0,32	3323

Table 3: Threads statistics for a poorly covered application.

ratio. Column *Average Activation Distance* gives the average time, in nano second, between two activations of the thread.

Table 2 gives the experiment data for the Powerstones DES benchmark. This example allows us to easily modify the input data size to check the robustness of the computed threads. Experiments show that there is no impact, due to the input data changes, on the data layout speculation computed. This observation meets A. Djabelkhir and A. Seznec [16] results on the behavior of embedded applications. However, we notice that the larger the data size is, the higher the thread 1 coverage is and the lower the thread 2 coverage is.

Table 3 presents the Powerstones adpcm benchmark. For this program, the average activation distance of the thread is very high. It would allow the system to shutdown the coprocessor between each usage.

7 Related Works

There are two main approaches for TLP. The first one requires a compiler intervention. The second one, is fully managed at hardware level. Due to hardware complexity, this second approach is usually limited to general purpose computing.

In the first approaches, the compiler is in charge of partitioning the application in threads. An hardware mechanism can then be used to deal with speculative data dependences or load

balancing. This is for instance the case with the *Expendable Split Window Paradigm* [17] and *Multiscalar* [18]. These mechanisms are well suited for shared memory systems and require complex runtime hardware mechanisms. Other, more static, approaches, such as PICO-NPA [11], SPSM [19], SuperThread [20] focus on loop parallelism without considering speculative execution. All data dependencies have to be resolved at compile-time, which limits the scope of these approaches for the exploration of application partitioning. A few works consider software only speculative thread parallelism [6, 21]. In this later case, the software is also in charge of checking data dependences at run-time. Some speedup was reported in Cintra's work [21]. The implementation of the threads can be improved using helper threads. These threads provide speedup by helping the prefetching of data, checking data dependence on the side, or to compute synchronization [22, 3, 23, 24, 25].

The second type of approaches proposed dynamic mechanism to detect and execute threads. No compiler intervention is needed. This is for instance the case for rePLay [5]. Hardware traces are used to compute speculative threads. If a thread is miss-speculated a hardware roll-back mechanism exploiting the underlying superscalar micro-architecture is used to restore a consistent state. Because of the hardware complexity of the approach, detected threads are usually limited in size. These approaches are not well suited for SoC especially with heterogeneous cores.

The approach proposed in this paper mainly focus on the partitioning at compile-time of the application while exploring the design space of a SoC. There is no hardware thread support assumptions.

8 Conclusion

In this paper we have presented a system able to automatically partition an application for an embedded system composed of a main CPU and a coprocessor. The system is based on speculative threads obtained from sequential C programs.

The proposed technique aims at helping the design exploration phase when decision must be made about which parts of the application to speedup using a coprocessor and what is the benefit and cost from such or such coprocessor. Preliminary experiments have shown that the speculative threads found are pertinent, and offer an interesting computation/communication ratio. Furthermore they are stable across data input sets.

Future work will focus on handling larger applications as well as integrating more static program analysis to reduce the amount of speculation. Reducing the amount of speculation will help the user to derive the effective implementation.

References

[1] Hans Zima and Barbara Chapman. *Supercompilers for parallel and vector computers.* ACM Press, New York, NY, USA, 1991.

[2] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2001.

[3] J. Oplinger and M. S. Lam. Enhancing software reliability with speculative threads. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 184–196. ACM Press, 2002.

[4] M. S. Lam J. T. Oplinger, D. L. Heine. In Search of Speculative Thread-Level Parallelism. In *PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, page 303. IEEE Computer Society, 1999.

[5] S. J. Patel and S. S. Lumetta. rePLay: A Hardware Framework for Dynamic Optimization. *IEEE Trans. Comput.*, 50(6):590–608, 2001.

[6] M. Chen and K. Olukotun. TEST: a tracer for extracting speculative threads. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 301–312. IEEE Computer Society, 2003.

[7] D. Brash. The ARM Architecture Version 6 (ARMv6). *http://www.arm.com/pdfs/ARMv6_Architecture.pdf*.

[8] B. Sinharoy R. Kalla and J. M. Tendler. Ibm power5 chip: A dual-core multithreaded processor. volume 24, pages 40–47. IEEE Micro, 2004.

[9] E. J. Marinissen T. Nguyen S. K. Goel, K. Chiu and S. Oostdijk. Test infrastructure design for the nexperia home platform pnx8550 system chip. In *Design, Automation and Test in Europe Conference and Exhibition Designers' Forum (DATE'04)*, 2004.

[10] E. Kilgariff and R. Fernando. Chap. 30. In *GPU Gems II*, pages 471–492, 2005.

[11] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. R. Rau, D. Cronquist, and M. Sivaraman. PICO-NPA: High-Level Synthesis of Nonprogrammable Hardware Accelerators. *J. VLSI Signal Process. Syst.*, 31(2):127–142, 2002.

[12] G. Pokam and F. Bodin. An Offline Approach for Whole-Program Paths Analysis using Suffix Arrays. In *LCPC '04: Languages and Compilers for Parallel Computing*, 2004.

[13] Gannon D. Bodin F., Beckman P. and Srinivas J.G.S. Sage++: A class library for building Fortran and C++ restructuring tools. In *Object-Oriented Numerics Conference*, 1994.

[14] Corinna G. Lee. UTDSP Benchmark.

[15] Motorola. Powerstone Benchmark.

[16] A. Djabelkhir and A. Seznec. Characterization of embedded applications for decoupled processor architecture. In *Proceedings of the IEEE 6th Annual Workshop on Workload Characterization*, 2003.

[17] Manoj Franklin and Gurindar S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelsim. In *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, pages 58–67, New York, NY, USA, 1992. ACM Press.

[18] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 521–532, New York, NY, USA, 1998. ACM Press.

[19] Pradeep K. Dubey, Kevin O'Brien, Kathryn M. O'Brien, and Charles Barton. Single-program speculative multithreading (spsm) architecture: compiler-assisted fine-grained multithreading. In *PACT '95: Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques*, pages 109–121, Manchester, UK, UK, 1995. IFIP Working Group on Algol.

[20] Jenn-Yuan Tsai, Zhenzhen Jiang, and Pen-Chung Yew. Compiler techniques for the superthreaded architectures. *Int. J. Parallel Program.*, 27(1):1–19, 1999.

[21] Marcelo Cintra and Diego R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 13–24, New York, NY, USA, 2003. ACM Press.

[22] Carlos Garcia Quinones, Carlos Madriles, Jesus Sanchez, Pedro Marcuello, Antonio Gonzalez, and Dean M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 269–279, New York, NY, USA, 2005. ACM Press.

[23] Yonghong Song, Spiros Kalogeropulos, and Partha Tirumalai. Design and implementation of a compiler framework for helper threading on multi-core processors. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 99–109, Washington, DC, USA, 2005. IEEE Computer Society.

[24] Yonghong Song, Spiros Kalogeropulos, and Partha Tirumalai. Design and implementation of a compiler framework for helper threading on multi-core processors. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 99–109, Washington, DC, USA, 2005. IEEE Computer Society.

[25] Jamison D. Collins, Dean M. Tullsen, Hong Wang, and John P. Shen. Dynamic speculative precomputation. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 306–317, Washington, DC, USA, 2001. IEEE Computer Society.

Contents