ASTEX: a Hot path Based Thread Extractor for Distributed Memory System on a Chip

Eric Petit, Guillaume Papaure, Florence Fdru@irisa.Fr Dru, François Bodin

HAL Id: hal-00418708

https://hal.science/hal-00418708

Submitted on 21 Sep 2009

ASTEX: a Hot path Based Thread Extractor for Distributed Memory System on a Chip

Eric Petit, Guillaume Papaure, Florence Dru and François Bodin

Irisa, Campus de Beaulieu, 35042 Rennes
`epetit@irisa.fr, bodin@irisa.fr`,
`http://www.irisa.fr/caps`

Abstract. This work addresses the problem of partitioning C code into threads at compile time in order to map them onto System on a chip (SoC). The Automatic Speculative Thread EXtractor, ASTEX, construct sets of thread partitions based on hot path. ASTEX 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 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.

1   Introduction

Most current embedded systems are System on a Chip (SoC) 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 partitioning process extracts threads that are distributed on the computing units of the system. One issue that SoC designers face is the exploration of the possible partition space to find a tradeoff between performance and system cost. Speeding up the exploration requires to built new automatic tools that compute potential partitions and qualitatively and quantitatively characterizes them. To fully address SoC it is also imperative to deal with distributed memory space which are often chosen for SoC since shared memory implies more complex design, higher energy consumption and performance penalty. In most cases, the input of the partition process is a sequential C program. Due to pointer aliasing issues, these programs are difficult to analyze statically. As a consequence, automatic static 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 propose ASTEX, a program for detecting threads in C programs. This can be used as a start-point for the partitioning of embedded applications for SoC. In ASTEX a thread is a set of hot paths with the corresponding memory working sets. Figure 1 illustrates the steps involved in the processing of a sequential application. This work focuses on the exploration phase which is divided in four main steps. The "potential thread detection" step finds the computation intensive phases named hot paths. The "Potential Thread Code Instrumentation" 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 accurate computation load of the candidate threads. The last step, according to execution results, evaluates the characteristics of the selected threads. The threads considered during this phase are speculative since they are computed based on runtime data. In ASTEX the speculative threads are explicitly built so the designer can exercice against various data inputs the chosen hot path to be converted into thread.
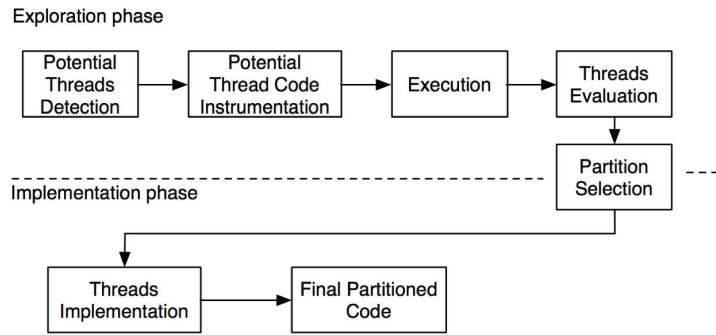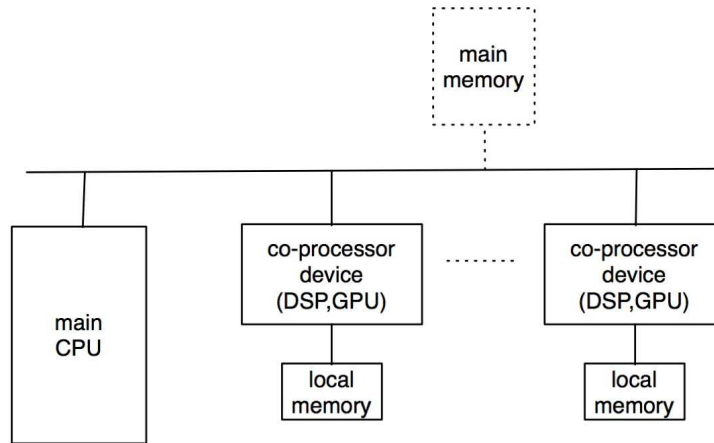
Fig. 1. Partition steps.

Fig. 2. Target Architecture Structure.

The selection step and the implementation phase may be performed with the help of the user. In many cases selection and implementation can also be automatized with the help of the information provided by ASTEX.

ASTEX is based on run-time data, is fully automatic and contrary to previous works [3–6], considers 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 ASTEX pipeline. Section 5 discusses the thread evaluation criteria. Section 6 reports the preliminary experiments. In Section 7 we overview the related works. Section 8 concludes this paper.

2 Target Architecture Model

The target architecture model is described in Figure 2. A main processor runs the application. Some coprocessors, having their own local memory, are used to speedup some parts of the application. Typically, the main processor is a RISC

micro-architecture [7–9] while the coprocessors can be VLIW processors, FPGA or any specialized computing unit such as ASIC or GPU [10, 9, 11].

A SoC implementing such a configuration aims at exploiting each processing unit where it is the most efficient. Usually, the coprocessors run compute intensive parts of the application.

Exploiting such a device is based on partitioning the application program into threads and then inserting the communication and synchronization between them. 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

The extraction process starts with an hot path detection based on executions traces. The hot path are the most frequently taken sub-path in the control flow graph during program execution. Next step instruments memory usage to characterize the memory working set for each hot path. Hot paths with their associated working sets form the threads. Since the threads are computed based on a set of execution, there is no guaranty that the collected data are valid for all executions; speculation arises at two levels:

1. At control flow level: The thread is assumed to correspond to a set of paths in the execution of the application program. If the execution of the thread leaves the assumed set of paths, then the speculation fails.
2. At data layout level: The data structures used in the application must be mapped in the local memory of the coprocessor. If a data accessed by the thread is out of the speculated memory space, the thread returns with an error code. This is one of the key points of this study.

Knowing accessed data for each thread enables to infer all data dependencies between them. When the speculation fails the computation performed by the thread must be resumed on the main CPU. To efficiently evaluate all the features of the threads implies efficiently running many full tests on as many representative input data sets as possible. An example is shown in Figure 3. The thread is made of basic blocks BB1', BB2', BB3' which are copies of BB1, BB2, BB3 in which the data accesses have been modified according to 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 initial 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. Restart only from BB1 has been chosen to avoid large and complex recovery code which would add overheads. Miss speculation also happens if the thread code gets out of the data space it has been allocated. If the thread terminates correctly, the thread 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 all the memory segments accessed by the program. In the following 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.
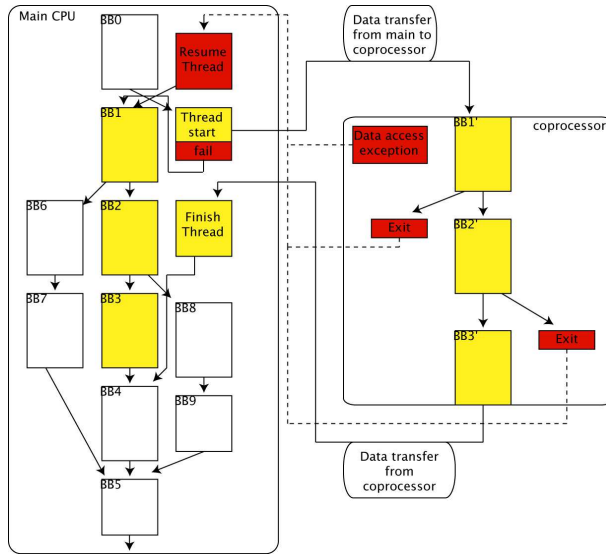
Fig. 3. Thread example.

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 sets* (AMS) that are defined by a unique *program creation point*, a *content type* , *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 to avoid data consistency problem inside the thread. 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*, *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 it is the end of the program.

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

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

The set of *abstract memory sets* available at a program point during the execution defines the *program working set* (PWS). An *abstract memory set* for which all memory blocks have been freed is removed from the PWS. An example of PWS is shown, 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 accessed 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
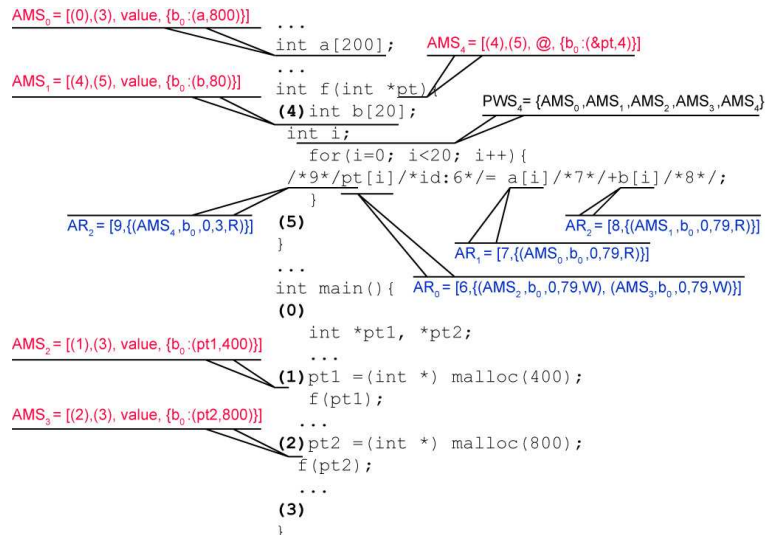
$AMS_0 = [(0),(3), value, \{b_0:(a,800)\}]$ ...

```
int a[200];
```
...

$AMS_4 = [(4),(5), @, \{b_0:(\&pt,4)\}]$

$AMS_1 = [(4),(5), value, \{b_0:(b,80)\}]$

```
int f(int *pt)
(4) int b[20];
    int i;
       for(i=0; i<20; i++){
    /*9*/pt[i]/*id:6*/= a[i]/*7*/+b[i]/*8*/;
       }
(5)
}
```

$PWS_4 = \{AMS_0, AMS_1, AMS_2, AMS_3, AMS_4\}$

$AR_2 = [9,\{(AMS_4, b_0, 0, 3, R)\}]$

$AR_2 = [8,\{(AMS_1, b_0, 0, 79, R)\}]$

$AR_1 = [7,\{(AMS_0, b_0, 0, 79, R)\}]$

```
...
int main(){
```
$AR_0 = [6,\{(AMS_2, b_0, 0, 79, W), (AMS_3, b_0, 0, 79, W)\}]$
```
(0)
    int *pt1, *pt2;
    ...
```

$AMS_2 = [(1),(3), value, \{b_0:(pt1,400)\}]$

```
(1) pt1 =(int *) malloc(400);
    f(pt1);
    ...
```

$AMS_3 = [(2),(3), value, \{b_0:(pt2,800)\}]$

```
(2) pt2 =(int *) malloc(800);
    f(pt2);
    ...
(3)
}
```

Fig. 4. Abstraction example.

memory block is determined. According to the block, the minimum and maximum offsets are computed. There is a unique AR 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 of AR are shown 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 usage of dynamically allocated memory.

3.3 Thread Memory Mapping

Let us assume that a set of basic blocks BB_1, ..., BB_n forms a hot path we wish to convert into thread. The memory elements used by the thread must be mapped onto the coprocessor unit. They are defined by the memory 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 of the data space 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 associates to an abstract memory set: an address, an offset and a memory size in the coprocessor local memory. The local memory is 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 corresponds to the amount of memory to copy (assuming the local space is large enough). Figure 5 shows an example of such mappings. It works as follow for abstract memory sets containing values (i.e. no addresses):

Global Arrays/Scalars: A memory space corresponding to a subset of the global variable is allocated onto the coprocessor memory. Only the subsets of elements really accessed by the abstract references are allocated.
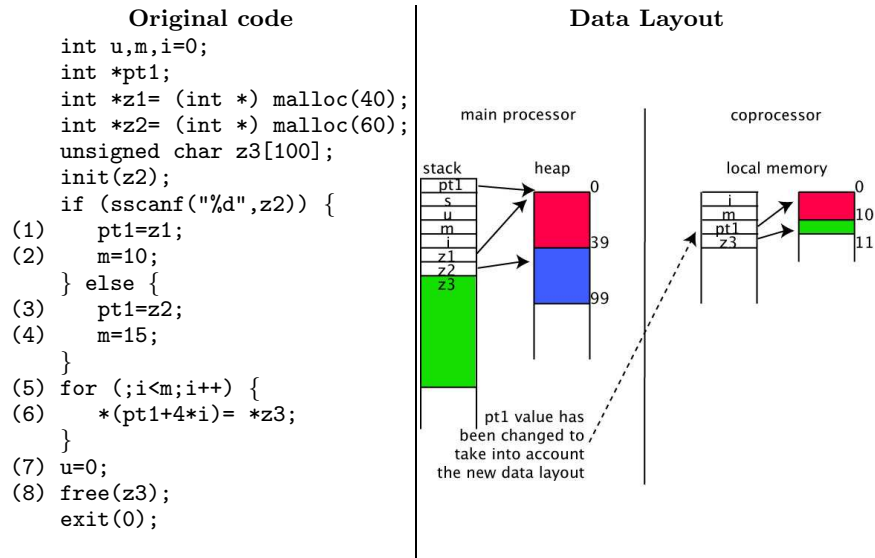
```
        Original code                          Data Layout
      int u,m,i=0;
      int *pt1;
      int *z1= (int *) malloc(40);
      int *z2= (int *) malloc(60);       main processor          coprocessor
      unsigned char z3[100];
      init(z2);                          stack    heap          local memory
      if (sscanf("%d",z2)) {              pt1                        i
(1)       pt1=z1;                          s    ┌──────┐ 0         m      ┌──────┐ 0
(2)       m=10;                            u    │      │           pt1    │      │ 10
      } else {                            m    │      │           z3     └──────┘ 11
(3)       pt1=z2;                          i    └──────┘ 39
(4)       m=15;                           z1   ┌──────┐
      }                                   z2   │      │
(5)  for (;i<m;i++) {                     z3   │      │
(6)      *(pt1+4*i)= *z3;                       │      │
      }                                          └──────┘ 99
(7)  u=0;
(8)  free(z3);                                pt1 value has
      exit(0);                              been changed to
                                            take into account
                                           the new data layout
```

Fig. 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.

Local Arrays/Scalars: A corresponding variable is allocated.
Dynamically allocated variables: For dynamically allocated variables, a local
array is created in the coprocessor memory.

The current scheme assumes all AMS in a thread are of type *value*. Otherwise it may imply indirect 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.

Scalar Pointer Variable Case is 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.

In order to implement an abstract memory set corresponding to a scalar pointer, a change in the address value is performed, so that an indirect access in thread is translated in the thread memory space.

Thread Guarded Memory Access Once the thread coprocessor memory space has been created the memory accesses must be guarded to ensure that they only addresses the abstract memory set that has been allocated onto the coprocessor. Miss speculation can happen 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.

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));
}
```

Fig. 6. Example of memory mapping table and guarded memory accesses. Integer scalar variables (`i, m, n`) are omitted from the table.

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));
}
```

Fig. 6. Example of memory mapping table and guarded memory accesses. Integer scalar variables (`i, m, n`) are omitted from the table.

The entries of the memory mapping table are statically computed and the value are filled at run-time. Figure 6 shows an example of guarded memory accesses (`check` function in the code). 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 ensures that there is no overflow of the coprocessor local memory. It should be noted here that the bound checking can easily be optimized in many cases by moving the tests outside the loops. The data speculation for expression $idex_2$ considers AMS_2 and AMS_3 which implies useless communications between 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 to compute the threads as well as when running the speculative threads. The history is limited in size but can be tuned for the applications. A too small history will induce more memory allocated on the coprocessor and more communications. A large history generates more execution run-time 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.

4 Extracting The Speculative Threads

In this section we overview the successive steps that ASTEX uses to build the speculative threads. One of the key point is to obtain an extraction pipeline with

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));
}
```

Fig. 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.

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 identifies 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 an execution trace. 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 the trace generated in step 2, computes a set of hot paths that will be the basis for choosing the code to execute on the coprocessor. The algorithm for computing the hot paths, has been proposed in [12] and is based on suffix arrays. Its 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 program 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 threshold), the blocks are collapsed into 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 threads are 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 ASTEX helps application designers choosing threads for a SoC.
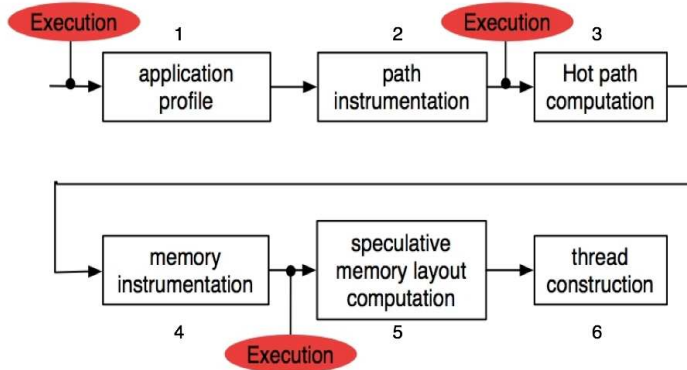

Fig. 8. Thread Extraction Overview.

5 Thread Selection

When threads have been identified and extracted many parameters may be analyzed during the selection step. First the *qualitative* analysis may be performed in order to ensure that the threads requirements meet by the coprocessor coprocessor. The *quantitative* analysis measure on input data sets the running properties of the threads.

5.1 Qualitative Analysis

The qualitative analysis studies 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, an ASIC or an FPGA, the requirements will not be identical.

5.2 Quantitative Analysis

This analysis studies 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 due to fork/commit function.
Communication: Since 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 arises, 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 knowing the target coprocessor architecture in order to be able to estimate the potential gain. An execution time gain, assuming no parallelism between the main program and the thread, 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. Since we explicitly construct the code of the thread, it is possible to estimate 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. Since 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 aim 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 benchmark suite [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. For each set of data we proceed in two steps to get the statistics. 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 partition. In average 59.7 % of execution time is covered by threads. The average communication volume at the activation of the thread is only 500 bytes. For all the applications the miss-speculation rate is small enough to be ignored (0 up to 10% on average).

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 lower than 30%.
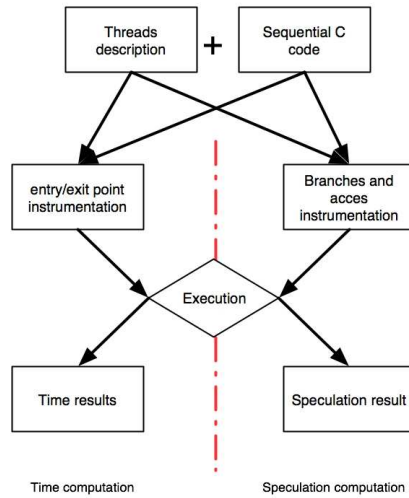
Fig. 9. Experimentation process

Tables 1, 2 and 3 report 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.

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.

Table 1 gives the results of the 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 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 ratio. Column *Average Activation Distance* gives the average time, in nano second, between two activations of the thread.

Table 2 gives the experimental data for the Powerstones `DES` benchmark. This example allows us to easily modify the input data size to check the speculation 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. Djabelkahir and A. Seznec [16] results on the behavior of
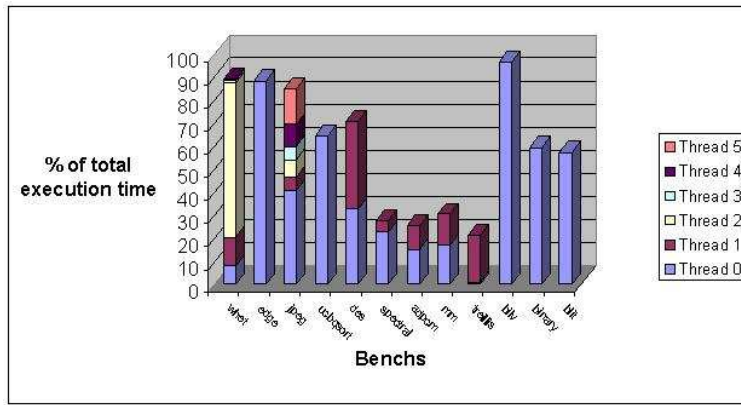
Fig. 10. Threads total coverage of execution time for all tested benches

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.

embedded applications. However, we notice that the larger the input data size is, the higher the thread 1 coverage is and the lower the thread 2 coverage is.

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.

Table 3 presents the Powerstones `adpcm` benchmark. For this program, the average activation distance of the thread is very high. For instance, this allows to shutdown the coprocessor between each usage.

7 Related Works

Our approach is related to what is usually called Thread Level Parallelism (TLP).

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 dependences on the side, or to compute synchronization [22, 3, 23–25].

The second type of approaches proposed dynamic hardware 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. Due to 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.

ASTEX 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. Contrary to most of the others approaches for threads extraction ASTEX assumes distributed memory systems with heterogeneous cores.

8   Conclusion

In this paper we have presented ASTEX, a system able to automatically generate threads partition of an application for an embedded system composed of a main CPU and coprocessors. 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 the coprocessors and what is the benefit and cost using such or such coprocessor. Preliminary experiments have shown that the speculative threads found in most of the tested benchs are pertinent in terms of computation and communication load. 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 cost and the amount of speculation. Reducing the amount of speculation will help the user to derive the final 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.