Code-Partitioning for a Concise Characterization of Programs for Decoupled Code Tuning

Eric Petit, François Bodin

To cite this version:

Eric Petit, François Bodin. Code-Partitioning for a Concise Characterization of Programs for Decoupled Code Tuning. 2010. hal-00460897v3

HAL Id: hal-00460897
https://hal.science/hal-00460897v3

Preprint submitted on 31 Mar 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Code-Partitioning for a Concise Characterization of Programs for Decoupled Code Tuning

Eric Petit and François Bodin

UPVD, Perpignan ; IRISA, Rennes
eric.petit@univ-perp.fr, bodin@irisa.fr,

Abstract. ASTEX, Automatic Speculative Thread EXtractor, is a compiler front-end we devellop for automatic C-code partitioning in speculative codelets, i.e. pure function that can be distributed over computing units of a system. It is an hybrid approach mixing static analysis and speculative data from profiles to offline generating a partition of the input C sequential code. In addition to the generated C partitioned application, we provide all the information on extracted kernels for a distributed memory heterogeneous system implementation and optimization. ASTEX makes a full abstraction of the target system for the programmer and provide useful guideline for code refactoring, if needed. It includes time profiling, communication, data layout information and data value prediction. Each codelet can be generated as a stand-alone program with associated input data set for effective iterative optimisation.
ASTEX results on NAS, SPEC 2006 and H264 benchmarks are relevant, and future works are on the way toward automatic specialized coprocessor programming.

1 Introduction

The growing demand of computing capabilities and energy efficiency have two consequences. On the one hand, there is an increasing parallelism with for instance multiplication of cores. On the other hand there is a specialization of cores. The resulting trend is heterogeneous many core with distributed memory or non-coherent shared one. Since most of the current applications, legacy codes and libraries are programed in sequential high level languages, all of them need to be rebuild for next generations of architecture.

It is necessary to develop languages, tools and optimizing-retargetable-compilers to deal with these new architectures and new softwares paradigm, including SPMD, TLP and stream computing model. Considering distributed memory system implies a strong memory analysis. It must deal with inter-thread communication, memory re-mapping between each local address space, and dependance analysis.

The first objective for compiling the application for heterogeneous-many-core is to find computing kernels and to map them efficiently on the different units. Since hand-writing distributed parallel program is a complex and time-consuming task for human programmers we want to assist them and in a long term to automatize parallel code generation.

Optimal code generation is an undecidable problem [1] and a high level language analysis with pointer aliasing for instance, drastically limits the static analysis to produce good results. The current known dynamic approaches for code partitioning with JIT extraction and parallelization have strong overheads, in hardware or software or both. Automatic off-line compilers are needed.

An effective way toward automatic partitioning can be found in hybrid approach mixing static analysis with speculative data from execution profile. It leads to an off-line code partitioning, reducing dynamic overhead to the minimum.

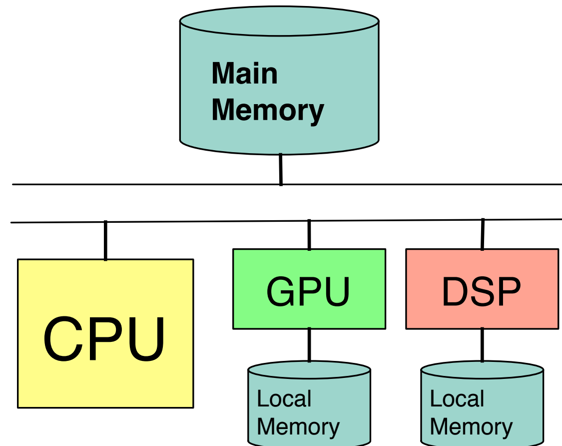

Fig. 1. Target Architecture Structure.

In this paper we present the ASTEX approach. It deals with the problem of heterogeneous distributed memory system efficient programming. It focuses on kernel detection, outlining and characterization. ASTEX results are a good starting point for hand-writing or automatic code tuning for heterogeneous distributed system programing, such as on-board hardware accelerator.

2 Paper Structure

This paper is organized as follow.

Section 3 is an introduction on distributed-memory heterogeneous system.

Section 4 introduces the state of the art for partitioning tools.

Section 5 describes our main contributions in heterogeneous system programming.

Section 6 starts our tool description.

Section 7 explains the computed results and how ASTEX produce them.

Section 8 shows experimental results.

Lastly we present the perspectives for future research and development based on the ASTEX approach.

3 Distributed-Memory Heterogeneous System

3.1 Distributed-Memory Heterogeneous Architecture

Figure 1 shows the paradigm of distributed heterogeneous architecture. A main processor runs the application. One or multiple coprocessors, having their own local memory, are used to speedup the processing of some parts of the application. Many systems match this model, for instance, specialized on board coprocessors such as GPU or clearspeed accelerators coupled with a general purpose multi-core.

Programming such an architecture implies to find kernel to execute out of the main CPU, to communicate data between processing units, and to schedule properly the execution. Because programing capabilities and ISA are very different from one core to another, programer should be able to mix multiple programming models and levels of parallelism (TLP, SPMD, ILP, SIMD, STREAM...)

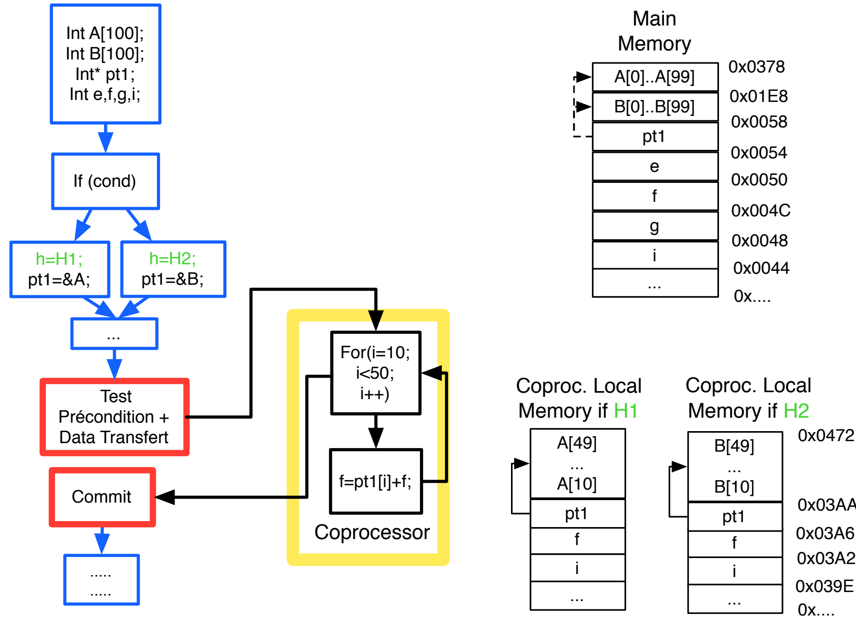

Fig. 2. Memory mapping example.

Distributed-memory heterogeneous architecture matches more and more systems, from embedded (SoC) to HPC (GPU grid). Consequently, there is a large number of various library and legacy codes to those architectures to port.

How efficiently map current applications onto this new kind of architecture?

3.2 Distributed-Memory Impact

This part focuses on problem linked with distributed memory. We assume there is no global address space.

Figure 2 shows the memory mapping needed while outsourcing a part of a Control Flow Graph, CFG, from a coprocessor with it's own local memory. The left part represents the CFG of an application. A subset of the vertices representing a loop are mapped onto a coprocessor. This loop computes an accumulation of elements 10 to 49 of the memory zone pointed by `pt1`. The representation of the memory mapping in main memory is on the top left of the figure. Since we are in a distributed memory system, data need to be mapped onto the local memory of the coprocessor before the kernel execution starts on it.

First of all we have to determine which data have to be remapped. For instance in figure 2, `pt1` can point onto arrays A or B of 100 elements each. In fact, for one call to kernel only a sub set from 10 to 49 of one of the two array is needed. The memory copied on coprocessor is represented on low right part of the figure. If previous execution goes through `H1`, only A from 10 to 49 is copied, else only `B` from 10 to 49 is copied. All pointers and associated arithmetic have to be converted in the local address space of the coprocessor. The value of pointer `pt1` has to be computed such as `pt1+10` points onto the first copied element of the array.

Distributed memory imposes complex communication (alias solving) and memory remapping.

4 Related Works

Software and hardware support for parallel computing are a long-lived field of research. The recent changes in computer architecture motivate a lot of new works around parallel application and specialized coprocessor usage such as GPGPU [19, 8, 14]. It can concern programming models [9], automatic code outlining [12], run-time management [13, 2, 17], and lastly hardware support [18, 22].

Astex is not the first tools to address the problem of code partitoning, but from our knowledge it is the first hardware independent one with the ability to handle distributed memory system and cores heterogeneity.

There are two main approaches for code partitioning. The first one requires a compiler intervention. The second one, is fully managed at run time mostly based on dedicated hardware mechanism. Due to hardware complexity, the second approach is usually limited to general purpose computing.

In static off-line approach, the compiler is in charge of partitioning the application in threads. Then hardware mechanisms can be used to deal with speculative data dependencies or load balancing. This is the case with the *Expendable Split Window Paradigm* [10] and *Multiscalar* [23]. These mechanisms are well suited for shared memory systems and require complex runtime hardware mechanisms. A few works consider software only speculative thread parallelism [5, 6]. In this case, the software is in charge of checking data dependencies at run-time. Some speedup was reported in Cintra's work [6]. 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 [3, 16, 24, 7].

The second type of approach proposed dynamic mechanism to detect and execute threads. No compiler intervention is needed. For instance this is the case for rePLay [18]. 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 focus on the partitioning at compile-time of the application. Astex use speculation to provide guidelines for decoupled code tuning since each kernel can be exercised separately. We makes no assumptions on hardware support. Contrary to most other state of the art approach, ours handles distributed memory.

5 Main Contributions

5.1 Speculative approach definition

Our approach consists on an off-line front-end for compiler. The input data for code generation is the source code of the application, and data from the execution profile. Therefore, the result is hybridism between pure static analysis and profile based dynamic information.

Since data are extracted from execution profile, their validity is limited to the execution context we observed. If we assume those information for all executions, they become speculative assumptions. Consequently, we must check the speculation validity at run-time.

Speculation arises at four levels:

1. At control flow level: A set of paths in the execution of the application program is assumed to be corresponding to the codelet. If the execution of the codelet leaves 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 the current study, this is not an issue since, we assumed there is no overlap between the execution of the main program and the threads (see section 3).
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 codelet returns with an error code. This is one of the key points of this study. The second way to speculate on data layout is to observe the alignment of access to data structures. We can specialized data layout to optimized access latency.
4. At scalar value level: If the input scalar value of codelet can be observed with a kind of stability, we can speculate on it, predicting their value.

Our approach differs from state of the art speculative approaches since speculation and associated tests are computed in early phase of the compilation and are architecture independent, i.e. no hardware mechanism. The consequences are the adaptability, the portability of the final generated source code and the possibility for each kernel to benefit from advanced optimization at compile time.

5.2 Execution Model

Figure 3 represents the execution model we defined for ASTEX. On the main processor is represented the CFG of a program. On the coprocessor is executed a codelet composed with Basic Blocks `BB1', BB2', BB3'` which are copies of `BB1, BB2, BB3`. In the codelet BB, data accesses have been modified according to the data mapping in the local memory (see section 3.2). Since the codelets are based on execution profiles, there is no warranty that they are valid for all executions; When the speculation fails the computation performed by the coprocessor must be resumed on the main CPU. In the partitioned program, if all pre-conditions are satisfied, the basic block `BB0` on figure 3 branches to a block that creates the job on coprocessor and copies the data in the coprocessor local memory. If the codelet flow of control leaves the speculated path `BB1', BB2', BB3'`, then the codelet is stopped and the main program resumes the execution at original block `BB1`. This also happens if the codelet working set gets out of the data space which has been allocated. When codelet finished, the non local modified data are copied back to the main memory.

Instead of waiting for codelet on the main processor, it is also possible to do some work, finding tasks to execute in parallel on main CPU. Since we focus on specialized coprocessor in a first step, we don't need this kind of parallelism to get speed-up. Considering a distributed memory system, we could start the same job as the coprocessor one on the main processor. In this case, if codelet fails on coprocessor, we just forget it and let the processor continue it's execution. Otherwise, Coprocessor commits its results and the main processor continues its execution from codelet return point. This is possible since codelet execution perfectly matches what should happen on the main processor. This policy is not work efficient but for small granularity codelet, it allows to remove most of the overheads from speculation failure.

6 The ASTEX Frame-Work

Figure 4 represents the ASTEX frame-work. It takes as an input a C sequential application and gives as a result a final application ready to be executed on the given architecture.

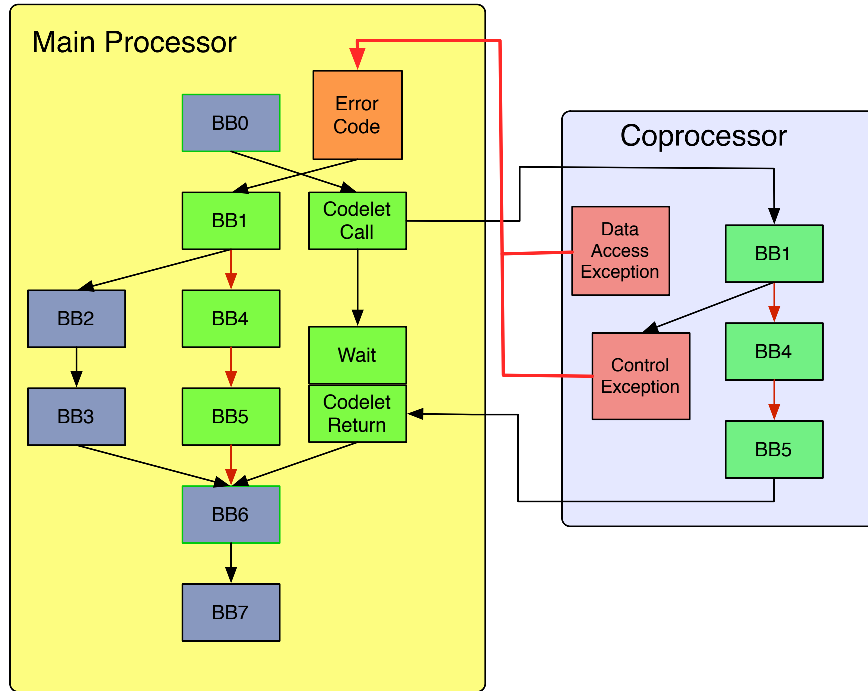

Fig. 3. Astex execution model.

The top right part of figure 4 represents Astex pipeline, Automatic Speculative Thread EXtractor. It consist on compiler front-end for automatic C-code partitioning in speculative codelets.

The low right part of figure 4 represents HMPP, Heterogeneous Manycore Parallel Programming [20]. It is a tool set which provides a pragma compiler and a set of architecture specialized back-end. It is an Open-MP-like dealing with heterogeneous distributed system. HMPP compiler aims at generating the needed version for all possible target coprocessor. At this point, back-end functionalities are still limited. Most of the time it is user responsibility to hand-write target specific code.

In this paper we focus on the top part of figure 4. The fully functional global frame work is our long-term objective.

7 Astex Features

Astex code-partitionner can be represented as a pipeline and indeed can be divided in stages. Each stage gives useful information for code partitioning, but also for code testing, tuning and refactoring.

The first step is candidate kernel detection for codeletization. The source code is instrumented, compiled and then executed. This generate an execution trace on which are computed hotspots. Detailed are given in section 7.1.

After codelet construction, i.e. automatic codelet code generation, we do a second step of instrumentation to detect memory usage in terms of communications, scalar values and memory zone alignments. All those informations are extracted using instrumented-application execution trace dynamically mapped onto an abstract memory model. Detailed are given in section 7.3.

In the following of this section, we detailed the different Astex pipeline stages and the associate features in output.

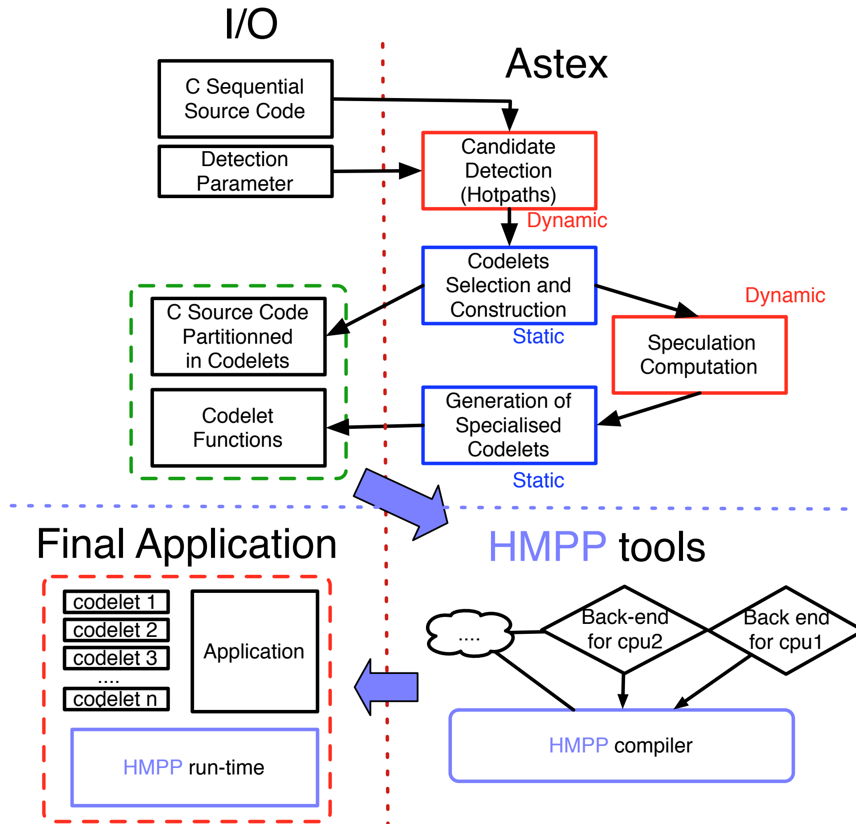

Fig. 4. Schematic view of Astex frame-work. Sequential application in C-source code are partitioned in "codelets" and then specialized, optimized and build for the host system.

7.1 Kernel Detection in Astex

The Kernel detection in Astex is based on hotpath, i.e. the most frequently executed path in the execution trace. This is done in accordance with Amdahl's law: the higher is the ratio of computation time you optimized the higher is the final speed-up. Most of the current parallelizing tool used nested-loops structure as a starting point for code partitioning, assuming an equivalence between loops and hotspots. In Astex, there is no assumption on code structure for kernel.

To detect and characterize the hotpath, we use a home-made specialized and optimized version of the tool based on G. Pokam work [21]. It is an implementation of the KMR algorithm [11] for rapid identification of repeated patterns in strings. It uses suffix arrays.

Since kernels are identified, they are outlined in function. This supposes a semantic completion of the instructions in kernel to match a code snippet that can be considered as a pure function (one entry, one exit, no side effect). Since we observed there are no immediate benefits to gain, we remove codelets internal control speculation. Therefore, hotpaths become hotspots. We use the terminology codelet to designate this outlined function, its memory usage profile and all specialized versions.

During the outlining phase, we do a first step of selection, removing unsupported semantic (external call, direct usage of pointer value...). When snippets have become codelets, we profile their execution time to determine if they are true hotspots. If

not, we remove them from the partition. This can happen since hotpaths coverages are computed from basic blocs number of occurrence and estimated cost in the global execution trace. To avoid false negative in codelet detection, the coverage ratio parameter is lowered for detection phase and fix to user-defined value for filtering.

During the detection phase appears the first level of speculation which arises on hotspots location and coverage .

7.2 Speculative Data Layout Representation

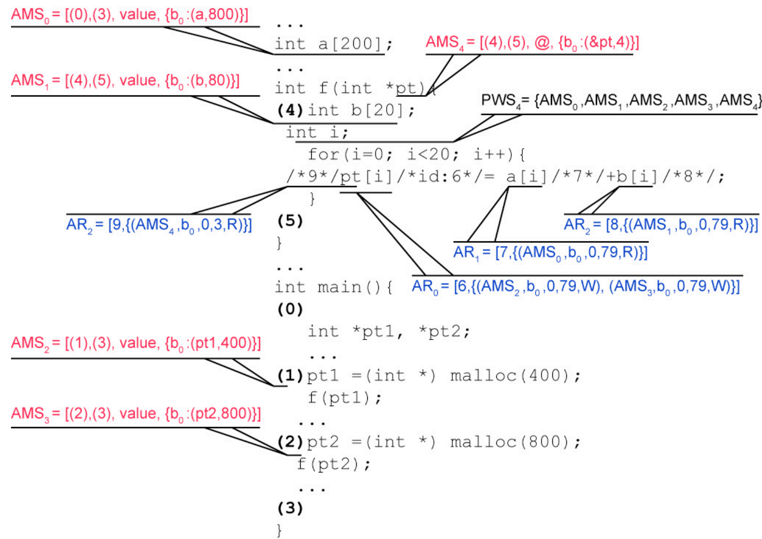

Fig. 5. Abstraction example.

The second step in ASTEX is memory usage speculation computation. While executing an instrumented version of the application with codelet, we mapped the collected information onto an abstract memory model describes in section 7.2.1 and 7.2.2. Those information are used for codelets and communications optimization and specialization as explain in section 7.3

7.2.1 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; a block can only belong to one AMS. This is a pre-condition to the codelet 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 the exit points for a given call site. The blocks corresponding to stack variables are also stacked in the abstract memory set.

For the codelet construction, only set containing scalar values (i.e. no complex data structures) are considered. Examples are shown, in red, in Figure 5.

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 5.

7.2.2 Memory Access Description

An *access* to memory, denoted *(@,R|W, s)*, is defined by the address (@), the access mode, read or write, ($R|W$), 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 memory block corresponding to address @ 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 5.

7.3 Speculative Specialization of Codelets

This section details the actual state of speculation for codelet specialization in Astex. Data layout and value speculation gives two ways of optimization:

- Communication optimization: speculating the accessed data, we can asynchronously copy and copy back only a subset of the possible accessed data according to static analysis.
- Specialization of codelet with automatic guard generation: since we know scalar values or data alignment values, we can generate specialized version of the codelet integrating those values.

The Communication optimization consists on reducing to the minimal set the transferred dataset. This is done to minimize the communication overhead. Using the memory model given in section 7.2, we compute the subset of program working set that speculatively acceded by a codelet.

The second way for communication optimization is to upload data asynchronously as soon as possible [19]. This is a way to hide communication latency. For instance, last NVIDIA GPU processor allow asynchronous transfer. In HMPP, this can be expressed through advanced load directives. The symmetrical problem, asynchronous downloading of result, uses delegated store directives. The problem of automatic communication placement optimization is a complex problem [4, 25, 19]. Speculative data can be really useful to compute good solution for communication generation [19]. It allows to speculate on the data dependencies (placing the upload after the last speculated write access), and on the communication content (minimizing the communicated set).

Scalar value prediction and data alignment specialization are another way for speculation usage. For scalar value prediction, we replace the value in the codelet

and generate a guard to call before codelet start. For data access alignments, the information are given through HMPP compiler directive. The experiments in section 8 show that stable scalar values and alignments are frequent.

7.4 Codelets Outlining

This section presents the codelet construction process from hotpath to specialized codelet function. We illustrate this with a small kernel example from h264, mpeg4 encoding/decoding application.

The hotpath we detect is in this example limited to one block of one instruction:

```
{
    i_sum += abs(pix1[x] - pix2[x]);
}
```

The first step is to extend the hotpath to immediate enclosing control statement, it means statement that includes only control statement including hotpath or other non-control statement. The result is the following:

```
for (y = 0 ; y < 8 ; y++)
{
    for (x = 0 ; x < 8 ; x++)
    {
       i_sum += abs(pix1[x] - pix2[x]);
    }
    pix1 += i_stride_pix1;
    pix2 += i_stride_pix2;
}
```

It is now possible to outline this snippet as a pure function. A simple static analysis gives live-in and live-out parameters and local variable. The resulting function is the following:

```
void Astex_codelet__1(uint8_t *pix1, int i_stride_pix1,
    uint8_t *pix2, int i_stride_pix2, int __Astex_addr__i_sum[1])
{
    int y;
    int x;
    int i_sum = __Astex_addr__i_sum[0];
    Astex_thread_begin: {
       for (y = 0 ; y < 8 ; y++)
       {
          for (x = 0 ; x < 8 ; x++)
          {
             i_sum += abs(pix1[x] - pix2[x]);
          }
          pix1 += i_stride_pix1;
          pix2 += i_stride_pix2;
       }  }
    Astex_thread_end:;
    __Astex_addr__i_sum[0] = i_sum;
}
```

After filtering on time coverage, the remaining codelets are instrumented to analyze their memory usage. After execution, we have some new information to integrate in the codelet. Since they are speculative, we obtained speculative specialized codelet. (see section 7.3) In the small example of this section, we observed that the value of `i_stride_pix1` is constant and equal to 16. Then it is possible to generate a specialized version of the codelet with this value:

```
void Astex_codelet__1(uint8_t *pix1, int i_stride_pix1, ...)
    { ...
          pix1 += 16;
          pix2 += i_stride_pix2;
    ... }
```

Since the generated codelet uses speculative data, we have to generate also precondition test, called *guard*, to ensure the speculated data are the actual one in the dynamic calling context:

```
    inline int Astex_guard__1(uint8_t *pix1, int i_stride_pix1,
    uint8_t *pix2, int i_stride_pix2, int __Astex_addr__i_sum[1])
    { ...
       return  i_stride_pix1 == 16 & ...
    ... }
```

Finally we integrate the codelet call in the main application. If the guard is true then the codelet call happen, otherwise the default code application is executed. Here is the example codelet call site in the main application:

```
    if Astex_guard__1(pix1, i_stride_pix1, pix2,
    i_stride_pix2, __Astex_addr__i_sum[1])
    {
       #pragma hmpp astex_codelet__1 callsite
       void Astex_codelet__1(pix1, i_stride_pix1, pix2,
       i_stride_pix2, __Astex_addr__i_sum[1])
    }
    else
    {
          for (y = 0 ; y < 8 ; y++)
          {
             for (x = 0 ; x < 8 ; x++)
             {
                i_sum += abs(pix1[x] - pix2[x]);
             }
             pix1 += i_stride_pix1;
             pix2 += i_stride_pix2;
          } } }
```

If needed, bound checking and control check are inserted in codelet code to ensure speculated data that are not live-in at codelet call.

7.5 Stand-Alone Codelet Generation

For iterative optimization or to test extensively computation kernel, it is more practical to execute codelet as stand-alone program. It avoids the whole program

execution. In order to extract the codelet, we need not only the codelet function, but the input data working set also. We have to construct specific data set from original data set of the whole application.

The first following section presents the specific data set construction method. The second one explain how the stand alone codelet program is build.

7.5.1 Automatic Generation of Specific Data Sets

During the whole application execution with codelet, we capture, using Astex memory model, for a given occurrence of the codelet, the data layout and value of parameters. We encode them in a binary file.

For each codelet, one or more data sets are created for stand-alone execution.

We also capture the corresponding output data-set for each input. Indeed we can check the results given by the codelet computation. We compare the original results with the ones product by specialized codelet version on different architecture. This validation is a critical point for specialized coprocessor usage, especially for numerical application which are dependent on numerical precision.

In the long term, will be generated, from the different speculative values of the scalar variables and data layout, test sets for each possible specialization.

7.5.2 Stand-Alone Test Application

Since codelet parameters are clearly identify and their value captured, the stand-alone codelet test bench generation presents no technical difficulties.

The generated program is a single `main` function. It first opens the binary file of input parameter and allocate all memory for codelet parameter and fill it with proper value. Each value is associated with one or more (alias) codelet parameters.

Then the codelet call happens. The output value are captured and sum-up in a binary structure. This output value can be then compared to original one.

7.6 Programming Guidelines Output

At the end of Astex pipeline we have two kinds of results. The first is an in-codelet partitioned application's C source code. This code can be compiled and executed using HMPP compiler. If compiled without HMPP, since HMPP pragma are written as comment, the code can be compiled by any C compiler. Codelets will be considered like other functions.

The second output is a human readable statistics report of the application execution. It gives useful qualitative and quantitative information about codelets (which are computing kernels of the current application). The output data are the following:

- Global execution context information: application name, input data sets, compiler version, host machine for execution...
- The detected hotpaths listed by execution coverage.
- The codelets list and their time coverage.
- The list of specialized codelet, their time coverage and the speculation success rate.
- For each detected hotpath:
  - The Hotpath code, the codelet code, the guards and specialized codelets code.
  - The list of scalar variables with the most frequent values if any.
  - The list of accessed memory zone.
  - The list of data access alignments.

Those information are useful for programmers since it spots the application kernel, specifying their memory usage and spot the critic data alignment. The explicit error description for rejected codelets makes rewriting possible.

8 Results

The following experimental results are based on the execution of three families of C benchmarks:

- The NAS parallel benchmarks consist in parallel algorithms in C and openMP. We want to match ASTEX results with expected one.
- H264: it is a multimedia encoding/decoding application with complex control flow driven by user-input options. Since this is a challenging application for programers, it is essential for ASTEX to get relevant results on it.
- The SPEC2006 benchmarks: it is compose of a various application, from simple to complex and small to large. We only considered C code ones.

In section 8.1, we present results about partitioning process. Section 8.2 presents results about speculation.

8.1 Codelet Partitioning

For all benchmark, the ASTEX execution time doesn't exceeding 7.33 times the original application execution time and evolved linearly with the treated application size. Furthermore, the dynamic allocated memory in the system while executing ASTEX pipeline on a benchmark never exceed 113 Mo (Mega-octet) and the execution traces a few Mo. This can be achieved with the dynamic compression of execution trace and the projection on the abstract memory model for data usage profiling. ASTEX is scalable for large application and practicable on desktop machine.

Figure 6 presents the cumulative time coverage of codelets in 22 applications of the three benchmarks suites. For H264, the coverage is stable around 60% whatever are the parameters, but codelets may be different. For NAS and SPEC2006, coverage are good enough for hardware accelerator usage (>40%) except for three applications. For the two worst of them, it comes from rejected codelets due to ASTEX software or model limitation. This can be overcome with kernel code refactoring. For the SPEC2006 one with 17% of coverage, there is no high density computing kernel in the application. The complexity is distributed along the whole application. For this kind of application, we can't find kernel for hardware accelerator.

Table 1 presents the statistic for success and rejection while computing codelets from hotpaths. Results are build on 417 codelets from the three benchmarks suites. 49% of the detected kernel become effective codelet in the partitioned application, 30% are rejected after codelet transformation since their real time coverage is too low (<1%) or they are included in another codelet. This kind of false detection is due to needed overestimation while computing the hotpath from program execution trace.

16% of the hotpath are rejected because ASTEX software doesn't match the whole theoretical model we designed. This last figure will be improve. The last 5% are the true reject, i.e. due to the model conception. They are mainly composed of hotpath using complex memory structure. This can't be avoiding without model improvement or code refactoring.

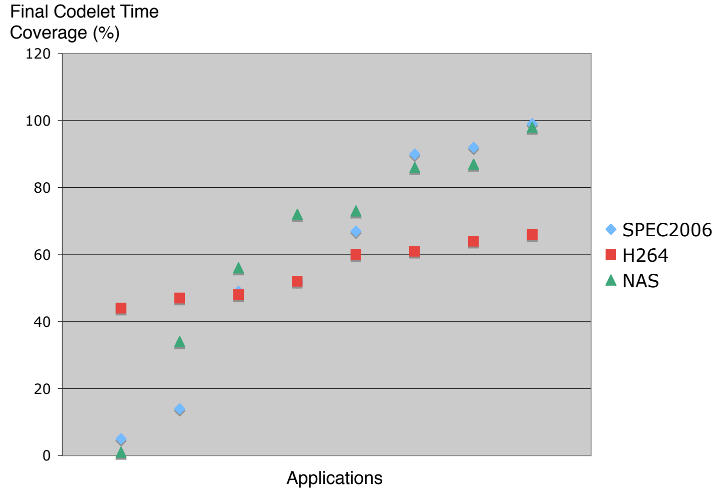

Fig. 6. Final codelet time coverage over the 22 tested benchmarks.

	Number of codelets	Rate in %
Success	205	49
Low Time Cov. (<1%)	123	30
Soft. Failure	68	16
Model Failure	21	5
Total	417	100

Table 1. Statistics for hotpath-codelet transformation results.

8.2 Speculative Specialization

Table 2 presents the statistic for speculative scalar-data value for all benchmark suites. Results are computed over 553 scalar variables. We consider as stable a scalar parameter with a unique value covering over 66% of the usage. We designate as *multi (2,3,4)* variables with 2,3 or 4 equiprobable values (e.g. A,B,C value for `alpha` variable with a uniform repartition is represented like this `alpha ={(A:33%),(B:33%),(C:33%)}`). *Stable* and *Multi* variables are predictable, i.e. we can use their value for specialization, and represent 58% of all variables. 18% are not live-in or live-out, indeed they are locale to the codelet and don't need communication, synchronization or specialization. Only 24% of all variables remain unpredictable and need communication. These results show a good potential for value prediction.

Table 3 presents speculative results for data access alignments. Results are computed over 905 access expressions from all the benchmark suites. *Multi* have the same definition as for scalar variable values. We consider as *stable* value that represents more than 85% of the observed alignments. *Stable* and *multi* are predictable and represent 91% of the case. Only 9% remains unpredictable because of instability. There is a high potential for success while speculating on data alignment.

	NAS	H264	SPEC2006	Total	Rate in %
Local	14	3	83	100	18
Stable (>66%)	34	166	53	253	46
Multi (2,3,4)	0	33	34	67	12
Instable	9	36	88	133	24
Total	70	241	278	553	100

Table 2. Statistics for data speculation on value . Three over four observed scalar values are predictable.

	NAS	H264	SPEC2006	Total	Rate in %
Stable (>85%)	204	126	357	687	76
Multi (2,3,4)	15	89	34	139	15
Instable	0	72	7	79	9
Total	219	287	399	905	100

Table 3. Data access alignments speculation. More than 90% of alignments are predictable.

9 Perspectives

Astex is a first step toward automatic partitioning of application. This first release gives a useful and powerful tool box for parallel programmers, decreasing the development time.

The next step is fully automatic code partitioning for legacy code for heterogeneous distributed memory system. The purpose is to optimize time-to-market for new processor designs by generating automatically library with good performance. To achieve this goal, we are currently working on back-ends for specialized coprocessor automatic code generation, based on Astex speculative profile.

A connected research field is in codelet/processor dynamic mapping using speculative profile of codelets and applications. This include affinity scheduler (between codelets themselves and between computing units and codelets), dynamic cost model (rentability evaluation before codelet call) but also concurrent execution of program and data management.

10 Conclusion

The automatic parallel code generation is an undecidable problem. There is numerous possible choice to make generating a huge solution space. Astex produces solutions, but we need tools such as iterative or learning compiler, to explore the solution space.

There is also a need for software and hardware normalization (e.g. OpenCL) [15] to guarantee success of automatic specialized code generation approach.

Astex introduces the concept of codelet, the associated formalization and the speculative execution model. Based on this model CAPS Entreprise develops HMPP and are then on the way to link it with Astex to get a full compiler frame work from sequential high level language program to distributed applications on heterogeneous systems.

The key point of our approach is speculation usage. It provides pertinent results for most of the applications. Astex computation time and memory usage are low and make the approach accessible to desktop computer.

It is already fully functional for parallel programming guidelines, outlining for decoupled code tuning , and stand-alone codelet generation for testing.

References

1. R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 63–76, New York, NY, USA, 1987. ACM.
2. C. Augonnet, S. Thibault, R. Namyst, and M. Nijhuis. Exploiting the cell/be architecture with the starpu unified runtime system. In *SAMOS '09: Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 329–339, Berlin, Heidelberg, 2009. Springer-Verlag.
3. n. Carlos García Qui C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. 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.
4. S. Chakrabarti, M. Gupta, and J.-D. Choi. Global communication analysis and optimization. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 68–78, New York, NY, USA, 1996. ACM.
5. 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.
6. M. Cintra and D. 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.
7. J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: long-range prefetching of delinquent loads. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 14–25, New York, NY, USA, 2001. ACM.
8. J. D. O. et al. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.
9. F.Bodin and R.Dolbeau. Hmpp programming tool home page. In *http://www.caps-entreprise.com/hmpp.html*.
10. M. Franklin and G. 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.
11. R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *STOC '72: Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 125–136, New York, NY, USA, 1972. ACM.
12. C. Liao, D. J. Quinlan, R. Vuduc, and T. Panas. Effective source-to-source outlining to support whole program empirical optimization. In *LCPC '09: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, 2009. ACM.
13. C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.
14. M. D. McCool, K. Wadleigh, B. Henderson, and H.-Y. Lin. Performance evaluation of gpus using the rapidmind development platform. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 181, New York, NY, USA, 2006. ACM.
15. A. Munshi. Opencl parallel computing on the gpu and cpu. In *SIGGRAPH'08*, 2008.
16. 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.

17. P. Palatin, Y. Lhuillier, and O. Temam. Capsule: Hardware-assisted parallel execution of component-based programs. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 247–258, Washington, DC, USA, 2006. IEEE Computer Society.
18. S. J. Patel and S. S. Lumetta. rePLay: A Hardware Framework for Dynamic Optimization. *IEEE Trans. Comput.*, 50(6):590–608, 2001.
19. E. Petit, F. Bodin, and R. Dolbeau. An hybrid data transfer optimization for gpu. In *Compilers for Parallel Computers (CPC2007)*, 2007.
20. E. Petit, F. Bodin, G. Papaure, and F. Dru. Astex: a hot path based thread extractor for distributed memory system on a chip. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 141, New York, NY, USA, 2006. ACM.
21. 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.
22. A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible decoupled transactional memory support. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 139–150, Washington, DC, USA, 2008. IEEE Computer Society.
23. G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 414–425, New York, NY, USA, 1995. ACM.
24. Y. Song, S. Kalogeropulos, and P. 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*, pages 99–109, Washington, DC, USA, 2005. IEEE Computer Society.
25. E. A. Stöhr and M. F. P. O'Boyle. A graph based approach to barrier synchronisation minimisation. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 156–163, New York, NY, USA, 1997. ACM.