Using OpenMP Superscalar for Parallelization of Embedded and Consumer Applications

Michael Andersch, Chi Ching Chi and Ben Juurlink
Embedded Systems Architecture Department
TU Berlin
10587 Berlin, Germany
Email: {andersch, cchi, juurlink}@cs.tu-berlin.de

***Abstract*—In the past years, research and industry have introduced several parallel programming models to simplify the development of parallel applications. A popular class among these models are *task-based* programming models which proclaim ease-of-use, portability, and high performance. A novel model in this class, OpenMP Superscalar, combines advanced features such as automated runtime dependency resolution, while maintaining simple pragma-based programming for C/C++. OpenMP Superscalar has proven to be effective in leveraging parallelism in HPC workloads. Embedded and consumer applications, however, are currently still mainly parallelized using traditional thread-based programming models. In this work, we investigate how effective OpenMP Superscalar is for embedded and consumer applications in terms of usability and performance. To determine the usability of OmpSs, we show in detail how to implement complex parallelization strategies such as ones used in parallel H.264 decoding. To evaluate the performance we created a collection of ten embedded and consumer benchmarks parallelized in both OmpSs and Pthreads.**

I. Introduction

Since the advent of multi-core processors and systems, programmers are faced with the challenge of exploiting thread-level parallelism (TLP). Writing parallel programs which exploit TLP is commonly considered to be difficult. To relieve programmers from the many issues associated with parallel programming, recently, several *parallel programming models* have been introduced, such as OpenMP [1], Boost Threads [2], POSIX threads [3], OpenCL [4], Intel ArBB [5], CUDA [6], OpenMP Superscalar [7], Cilk++ [8], and Intel TBB [9].

OpenMP Superscalar (OmpSs) is a novel task-based parallel programming model which extends the OpenMP programming model with the StarSs [10] task directives. In OmpSs, programs are parallelized by annotating functions as tasks using the `omp task input output inout` pragmas. When these annotated functions are called, they are added to a task graph instead of being executed. The task dependencies are resolved at runtime, using the input/output specification of the function arguments. Once all input dependencies of a task are resolved, it is scheduled for execution.

OmpSs has its origins in the HPC domain and previous evaluations of the OmpSs programming model have shown promising results [11]. The applicability of OmpSs to the embedded and consumer application domains, however, remains undiscussed. Applications in these domains are currently commonly parallelized using manual low-level threading approaches, which provide explicit control in the parallelization strategy. Other task-based programming models, such as OpenMP and Cilk++, have yet to be widely adopted in embedded and consumer application domains. Compared to HPC applications, there is usually less data-level parallelism available and the execution time is less dominated by small pieces of the application.

A key difference between OmpSs and other task-based parallel programming models is the ability to add tasks before they are ready to execute, a powerful feature which allows more complex parallelization strategies that simultaneously exploit function-level and data-level parallelism (DLP). In embedded and consumer application this is often required to compensate for the less abundant DLP.

In this paper we will investigate the usability and performance of OmpSs for embedded and consumer applications. The contributions of this paper can be summarized as follows:

- We propose an evaluation methodology for classifying and comparing the qualities and features of parallel programming models.
- We show how to express complex parallelization strategies in OmpSs using H.264 decoding as a case study.
- We discuss key usability aspects of OmpSs, such as the toolchain quality, debugging facilities, and adaptability.
- We evaluate the performance of OmpSs using ten embedded and consumer benchmarks, which are implemented in both OmpSs and Pthreads.

The structure of this paper is as follows: Section II discusses the methodology and the benchmark suite. In Section III we investigate the expressiveness of OmpSs using H.264 video decoding as a case study. Section IV extends this qualitative analysis by investigating other important usability aspects. In Section V the performance results of the ten benchmarks are presented and analyzed. Section VI discusses related work and, finally, in Section VII the conclusions are drawn.

II. Methodology

Evaluating parallel programming models is different from evaluating processor architectures. Parallel programming models not only target good performance, but also must offer the right abstraction to the programmer. Therefore, it is necessary to investigate both the usability and performance of a parallel programming model to evaluate its overall quality. To achieve

Name	Type	Application	Domain	Input Set	Code Size
c-ray	K	Offline Raytracing	Computer Graphics	192 objects, 1920×1080	500 LOC
md5	K	MD5 Calculation	Cryptography	512 MB	1000 LOC
rgbcmy	K	Color Conversion	Image Processing	30.5 Megapixels	700 LOC
rotate	K	Image Rotation	Image Processing	30.5 Megapixels	1000 LOC
kmeans	K	k-Means Clustering	Artificial Intelligence	18 dimensions, 2670 points, 500 centers	600 LOC
rot-cc	W	rotate + rgbcmy	Combined Workload	30.5 Megapixels	1400 LOC
ray-rot	W	c-ray + rotate	Combined Workload	192 objects, 1920×1080	1300 LOC
streamcluster	W	k-Median Clustering	Data Mining	1M points, 128 dimension, 10-20 centers	1100 LOC
bodytrack	A	Person Tracking	Computer Vision	4 cameras, 4000 points, 5 annealing layers	6800 LOC
h264dec	A	H.264 Decoding	Video Processing	3840×2160, YUV420, 50 fps	20000 LOC

TABLE I
EMBEDDED AND CONSUMER BENCHMARKS USED FOR THE PERFORMANCE EVALUATION. FOR EACH OF THE BENCHMARKS A SEQUENTIAL, PTHREADS, AND OMPSS VARIANT IS DEVELOPED.

these goals we will evaluate OmpSs using a set of qualitative usability aspects as well as a benchmark suite covering the embedded and consumer application domains.

A. Usability Aspects

The usability of a programming model is a subjective measure which differs from programmer to programmer. To give the programmer the necessary information to be able to form its own well-informed opinion about the usability, the following list of qualitative aspects of a programming model is discussed in this paper:

- Expressiveness
- Required code modifications
- Optimization opportunities
- Compilation toolchain
- Verification and debugging
- Code size

Because OmpSs is a novel programming model and currently not widespread, we found it is necessary to devote more attention to expressiveness, required code modifications, and optimization opportunities aspects. We, therefore, conduct a case study in Section III, presenting the parallelization of the H.264 decoding benchmark. This case study will show in detail how some of the more complex parallel programming constructs can be realized in OmpSs, without requiring excessive changes to the sequential base code. Additionally, we will show how OmpSs specific optimization applied to H.264 decoding effect the performance. The remaining qualitative aspects of a programming model are discussed in Section IV.

B. Benchmark Suite

To evaluate the performance of OmpSs, we have created a benchmark suite for the specific purpose of evaluating parallel programming models [12]. The suite contains 10 C/C++ benchmarks, partly taken from well-known benchmark suites as PARSEC and listed in Table I, covering a wide range of embedded and consumer application domains. The benchmarks are classified as kernels, workloads, and applications, based on their code size and parallelization complexity. For each benchmark a sequential, Pthreads, and OmpSs variant has been developed. For comparability the Pthreads and OmpSs variants exploit the same parallelism.

1) Kernels: In our benchmark suite, *kernels* are small programs with less than 1000 LOC. The OmpSs implementations exploit only DLP, using a single type of task without dependencies. The benchmarks mainly differ in their computation-to-communication ratios and the length of the parallel phases.

The *c-ray* kernel is a simple, brute-force ray tracer without post-processing. Each ray which is shot into the scene can be traced independently. To achieve a reasonably coarse granularity of work units, several rays must be grouped together. In the Pthreads implementation, each thread computes an equally sized horizontal band of the image. In the OmpSs implementation, the processing of one full image scanline is annotated as a task.

The *md5* kernel calculates the MD5 checksum of several independent input buffers simultaneously. Therefore, parallelism is abundantly available. For each input buffer, a separate task/thread is created.

The *rotate* and *rgbcmy* kernels perform image processing operations (rotation and color conversion, respectively). Both kernels are pixel independent and mainly differ in their memory access pattern. As in c-ray, the granularity must be coarsened to achieve good performance. Therefore, processing of a block of several image scanlines is performed by a task/thread.

The *kmeans* kernels solves the problem of clustering an arbitrary amount of n-dimensional data points using a k-means offline clustering algorithm. The computationally most demanding part is the calculation of the distance of each input point to every current cluster center. This computation can be performed independently for each input point, yielding a straightforward parallelization approach by executing blocks of input points in threads/tasks.

2) Workloads: Workloads are medium size programs (1000-5000 LOC) with 2 or more types of tasks and medium complexity parallelization. The suite contains the *rot-cc*, *ray-rot*, and *streamcluster* workloads.

The rotate + color convert (*rot-cc*) and the c-ray + rotate (*ray-rot*) workloads are derived by chaining the two kernels after another. In both cases, the output from the first kernel is fed into the input of the second kernel, yielding simple inter-task data dependencies.

The *streamcluster* workload originates from the PARSEC

2.1 benchmark suite [13]. Like kmeans, it solves the clustering problem. There are two differences: First, streamcluster uses an on-line method, processing a continuous stream of input data, and second, it employs a state-of-the-art k-median clustering algorithm. Parallelism is exploited in several phases of the algorithm, but is always based on the independence of the input data points. The parallel phases differ, however, in their duration.

3) Applications: Applications are larger programs with many tasks and different parallel stages. The benchmark suite contains two applications, *bodytrack* and *h264dec*.

The *bodytrack* applications originates also from the PARSEC 2.1 benchmark suite. Bodytrack tracks the 3D pose of a marker-less human body through an image sequence generated with four cameras with different view points. In the benchmark, a frame is read from each camera sequence and an edge map and binary map are created from it (preprocess), followed by a configurable number of annealing iterations estimating the 3D pose (estimation). Finally, the resulting pose is projected on the original input image of each camera and written to the disk (projection). The entire process is repeated for each frame in the sequence.

In the PARSEC implementation the parallelism is exploited in four parallel kernels, the edge map edge detect and edge smoothing kernels of the preprocess stage, and the particle weight calculation and particle resampling kernels of the estimation stage. The PARSEC implementation, however, exhibits limited scalability. After carefully studying the characteristics, we decided to reimplement bodytrack with a more scalable parallelization strategy. In our parallelization strategy the three stages are pipelined on the frame level, which reduced the sequential part of the application. Additionally, within the preprocess and projection stage parallelism is exploited on the camera level and the particle resampling and particle weight calculation of the estimation stage are fused in one pass, thereby reducing the parallel overhead by increasing the work unit sizes.

The *h264dec* application is a high performance parallel H.264 decoder. The decoder originates from the highly popular FFmpeg transcoder [14], which contains encoders and decoders for a plethora of audio and video codecs. The *h264dec* is extracted from the FFmpeg transcoder and has previously been parallelized using Pthreads. The H.264 decoder is compliant to the High profile Level 5.1, which enforces one slice per frame. The parallelization strategy is highly scalable by exploiting both FLP and DLP, and is capable of decoding QFHD (3840x2160) resolution sequences in realtime. This benchmark will be discussed in detail in Section III, where it serves as the OmpSs implementation case study.

C. Experimental Setup

To provide meaningful results not only for contemporary, but also for future multi-core systems, it is necessary to extend the benchmarking process beyond the core counts of what current off-the-shelve CMPs can offer. To achieve this, we use a 4-socket cc-NUMA machine with a total of 32 cores for the performance evaluation. The full hardware and software specification of our evaluation platform is listed in Table II.

Hardware		Software	
Processor	Xeon X7550	OS	Ubuntu 10.10
Cores	8	Kernel	2.6.35.10
Frequency	2.00 GHz	Compiler	GCC 4.4.5
Last level cache	18 MB	OmpSs compiler	Mercurium (git Aug'11)
Sockets	4		
Total cores	32	OmpSs runtime	Nanos++ (git Aug'11)
Total memory	1 TiB		
Total mem. BW	102.3 GB/s	Opt. level	-O2
SMT	Disabled		
Turbo mode	Disabled		

TABLE II
EXPERIMENTAL SETUP.

III. CASE STUDY: PARALLELIZING H.264 DECODING IN OMPSS

In embedded and consumer applications the parallelism is less abundant compared to, for instance, HPC applications. To achieve good speedup and efficiency, more complex parallelization strategies are required. A good example is the challenging *h264dec* benchmark. It is challenging because it is large, there is no single kernel that dominates the execution time, and parallelism is not abundantly available. One, therefore, needs to consider the entire application and exploit both data-level parallelism (DLP) as well as function-level parallelism (FLP) to obtain a high-performance and scalable implementation. Furthermore, both the DLP and FLP exhibit non-trivial data dependencies. For these reasons, it is an excellent benchmark to evaluate the expressiveness, required code modifications, and optimizations opportunities of OmpSs.

A. Pipelining H.264

The H.264 decoder pipeline consists in our design of 5 pipeline stages, shown in Figure 1. In the read stage the bitstream is read from the disk and parsed into separated frames. In the parse stage the headers of the frame are parsed and a Picture Info entry in the Picture Info Buffer is allocated. The entropy decode (ED) stage performs a lossless decompression by extracting the syntax elements for each macroblock in the frame. Some syntax elements are directly processed, e.g. the motion vectors differences are transformed into motion vectors. The macroblock reconstruction stage allocates a picture in the Decoded Picture Buffer and reconstructs the picture using the syntax elements and motion vectors. The output stage reorders and outputs the decoded pictures either to an output file or the display.

In contrast to other task-based programming models, like Cilk++ and OpenMP, pipeline parallelism can be expressed in OmpSs, because OmpSs tasks can be spawned before its dependencies have been resolved [15], [16]. Listing 1 presents the simplified code of the pipelined main decoder loop using OmpSs pragmas. A task is created for each pipeline stage in each loop iteration. For correct pipelining of the tasks, it is required that all tasks in iteration i are executed in-order.

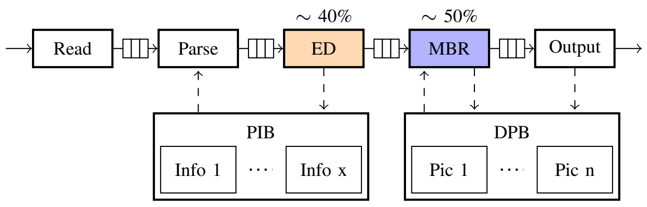


Fig. 1. H.264 decoder pipeline stages in our design

```
EncFrame frm[N]; Slice slice[N];
H264Mb* ed_bufs[N]; Picture pic[N];
int k=0;
while(!EOF){
    #pragma omp task inout(rc) output(frm[k%N])
    read_frame_task(rc,&frm[k%N]);
    #pragma omp task inout(nc,frm[k%N]) output(slice[k%N])
    parse_header_task(nc,&slice[k%N],&frm[k%N]);
    #pragma omp task inout(ec,slice[k%N]) input(frm[k%N])
        output(ed_bufs[k%N])
    entropy_decode_task(ec,&slice[k%N],&frm[k%N],ed_bufs[k%
        N]);
    #pragma omp task inout(rec) input(slice[k%N],ed_bufs[k%
        N])
        output(pic[k%N])
    reconstruct_task (rec,&slice[k%N],ed_bufs[k%N],&pic[k%N
        ]);
    #pragma omp task inout(dc) input(pic[k%N])
    display_task(dc,&pic[k%N]);
    k++;
    #pragma omp taskwait on (*rc)
}
```

Listing 1. Pipelining the main decoder loop using OmpSs pragmas

To accomplish this, each task in the same iterations is linked to the previous task in the same iteration via one or more input and output/inout pairs. Additionally, task *T* of iteration *i* must be completed before the instance of the same task *T* in iteration *i+1* is started. To accomplish this, each task has a context structure that is annoted as inout, e.g., ReadContext *rc, NalContext *nc, EntropyContext *ec, etc.

Three additional important observations regarding the pipelining implementation can be made. First, the `taskwait on` pragma ensures that the read task has been performed before evaluating the while loop condition. This is necessary to prevent tasks being added after the EOF has been reached.

Second, more importantly, the inputs and outputs of each task is using an entry of a circularly buffer of size *N*. This eliminates the WAR and WAW hazards that would have occurred if the same entry is used in each iteration, which would eliminate all the parallelism. OmpSs does not yet support automatic renaming and therefore, at the moment, this manual renaming method is required.

Third, the Picture Info Buffer (PIB) and Decoded Picture Buffer (DPB) structures are not passed in any argument and, thus, are not considered for dependence checking. To avoid having to reallocate the PictureInfo and DecodedPicture structures every iteration, the entries of the PIB and DPB are reused in the sequential code. The dependencies to these buffer entries are purposely hidden from the OmpSs task specifications, because we cannot predict which buffers entries will be available at the time the task is spawned. This can only be determined when the task is executed.

To fetch and release the buffer entries in a thread-safe way, `omp critical` pragmas are used in the task bodies around the fetch and release statements to protect accesses to the PIB and DPB. For this method to work there must be enough buffer entries available to accommodate the maximum parallelism. Due to the manual renaming this is controlled by the variable `N`, which leads to `N+16` buffer entries for the PIB and DPB to also accommodate for 16 reference frames.

Exploiting the FLP, however, is not enough to get scalable performance. The performance of the pipelined implementation is limited by the longest stage in the pipeline. The entropy decode and macroblock reconstruct stages take around 40% and 50% of the total execution time, respectively. The total speedup is in this case limited to a factor of two. To gain additional speedups both the entropy decode and macroblock reconstruct stages must be further parallelized.

B. *Parallelizing Entropy Decoding*

The ED stage performs entropy decoding using CABAC or CAVLC. In both these methods the interpretation of each bit in the stream depends on the previous bit. Therefore, no task parallelism exists inside the entropy decode of one frame. Multiple frames, however, can be decoded in parallel as they are separated by start codes. The frames, however, are not fully independent as illustrated in Figure 2.

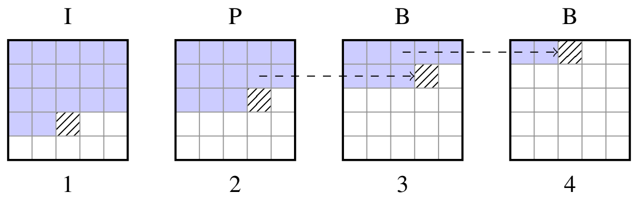


Fig. 2. Parallelism in entropy decoding in multiple consecutive frames. Colored MBs have been entropy decoded. Hatched blocks are decoded in parallel.

In Figure 2, four frames are decoded in parallel. The hatched blocks represent the current MBs that are decoded in parallel and the blue blocks denote the already processed MBs in each frame. For blocks in the B-frames some blocks may need the motion vectors of the co-located block in the previous frame. To express this parallelism, the code segment in Listing 2 can be used replacing the entropy_decode_task in Listing 1.

The entropy decode task is split in three tasks. The init task initializes the context tables, the decode_entropy_line_task entropy decodes a macroblock line of the picture, and the release task releases one or more Picture Info entries, which are no longer referenced. To have multiple entropy decodes in flight, the EntropyContext is renamed in the same fashion as the pipeline buffers shown in Listing 1.

To maintain the dependencies shown in Figure 2, the entropy_decode_line_task has several annotated arguments. The EntropyContext is annotated with inout to enforce that the lines of each picture are decoded sequentially. H264Mb

```
...
#pragma omp task inout(ec[k%N], slice[k%N], ed_bufs[k%N])
     input(frm[k%N])
init_entropy_task(&ec[k%N], &slice[k%N], &frm[k%N],
  ed_bufs[k%N]);
for(int i=0; i<row; i++){
  #pragma omp task inout(ec[k%N], slice[k%N]) input(ed_bufs
       [(k+N-1)%N][i])
  output(ed_bufs[k%N][i])
  entropy_decode__line_task(&ec[k%N], i, &slice[k%N],
    &ed_bufs[(k+N-1)%N][i], &ed_bufs[k%N][i]);
}
#pragma omp task inout(slice[k%N], k)
release_PI_task(&slice[k%N], &k);
...
```

Listing 2. Code fragment replacing the entropy task to perform parallel entropy decoding.

*mb_in is annotated as an input and H264Mb *mb_out is annotated as an output, to maintain the dependencies between frames. By passing the pointers to the first H264Mb of the co-located line in the previous entropy buffer and the first H264Mb of the current line in the current entropy buffer to mb_in and mb_out, respectively, it is ensured that each line *x* in frame *n* is decoded before starting to decode line *x* in frame *n+1*. A task for each macroblock line is created instead for each macroblock to increase the task granularity at expense of parallelism. The parallelism is still sufficient, however, with a maximum of 135 for QFHD resolution videos.

While the code fragment in Listing 2 is able to correctly process multiple frames in parallel, it is not able to take advantage of the fact that I- and P-frames do not depend on the previous frames, as depicted in Figure 2. At the time the ED tasks are added the frame type is not known, and, therefore, it must be assumed to have the worst case dependencies for correctness.

C. Parallelizing Macroblock Reconstruction

In the macroblock reconstruction (MBR) stage the image is reconstructed using the syntax elements produced by the entropy decoding stage. To reconstruct a macroblock in H.264 several pixel areas from adjacent reconstructed macroblock are required. For each hatched macroblock in Figure 3, the adjacent red pixels are needed for the intra-prediction and the deblocking filter kernels. Therefore, only macroblocks on a wavefront are parallel. The wavefront parallelism is not massive, but sufficient with a maximum of 120 free macroblocks in QFHD resolution videos. The wavefront parallelism can be exploited using the code fragment in Listing 3.

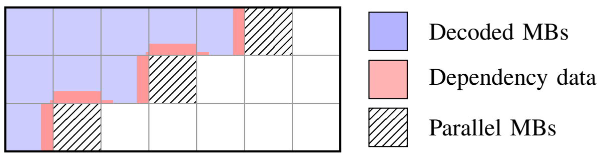


Fig. 3. Wavefront parallelism in H.264 macroblock reconstruction.

The wavefront dependencies are static and are covered by the dependencies to the left macroblock and the upper right macroblock. The wavefront dependencies of the reconstruct_mb_task are maintained through its H264mb*arguments by annotating the left macroblock *ml* and upper right macroblock *mur* as inputs and the current macroblock *m* as inout. Since the reconstruct_mb_tasks are added in scan line order, this input and the output specification ensures that the wavefront dependencies are maintained. In addition to the pragmas, only the code that computes the left and upper right macroblock must be added to the sequential code. The ability to express dependencies between tasks makes the OmpSs implementation relatively clean and simple.

```
#pragma omp task input(*rc, *s, *ml, *mur) inout(*m)
void reconstruct_mb_task(MBRecContext *rc, Slice *s,
  H264mb *ml, H264mb *mur, H264mb *m);

#pragma omp task inout(*rc) input(*s, mbs[0;rows*cols])
  output(*pic)
void reconstruct_task(MBRecContext *rc, Slice *s,
  H264Mb *mbs, Picture *pic){
  init_ref_list(s);
  get_picture_buffer(rc, s);
  for(int i=0; i< rows; i++){
    for(int j=0; j< cols; j++){
      H264mb *m = &mbs[i*cols + j];
      H264mb *ml = m - ((j > 0) ? 1: 0);
      H264mb *mur = m - (((j < cols-1) && (i >0))
        ? cols-1: 0);
      reconstruct_mb_task(rc, s, ml, mur, m);
    }
  }
  H264mb *lastmb = &mbs[smb_width*smb_height -1];
#pragma omp taskwait on (*lastmb)
  release_ref(rc, s);
  *pic = s->pic;
}
```

Listing 3. Wavefront algorithm expressed in OmpSs.

D. Optimizing Task Granularity

The OmpSs implementation of parallel macroblock reconstruction shown in Listing 3 is a clean way to express the wavefront parallelism. The decode_mb_tasks, however, are fine-grained and have an average execution time of around $2\mu s$ on a commodity processor. The task management overhead of OmpSs does not allow such fine-grained tasks to perform well. A technique to overcome this is to coarsen the tasks by grouping several macroblocks. Due to the wavefront dependencies, however, macroblocks must be grouped in tetris-block shapes, as shown in Figure 4.

Expressing task dependencies between these tetris-shaped superblocks directly is not straightforward, especially when it is desired to support an arbitrary picture and superblock sizes. To overcome this the superblocks are remapped to a regular structure. Independent of their shapes, all superblock variants still exhibit wavefront dependencies. The dependencies can be easily checked, in the same way as the regular ungrouped macroblocks when they are remapped to a regular matrix form as depicted in Figure 4.

The remapping does somewhat increase the code complexity. First, it must be calculated how many superblocks fit in the width and the height of a picture. Second, when decoding the superblock it must be checked which macroblocks belong

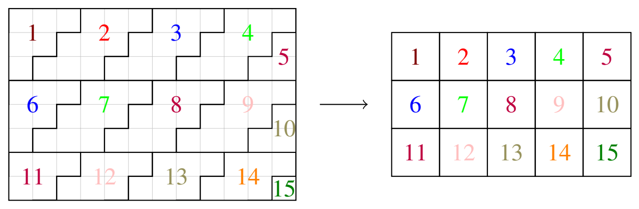


Fig. 4. Remapping the irregular superblock shapes back to regular shapes simplifies the dependency expression for the programmer and task dependence checking for the runtime system.

to this particular superblock. This check can be performed by testing if the macroblocks covered by the superblock shape are inside the picture.

If it is ensured that every first superblock in a line has the maximum number of macroblocks in its top row, as is the case in Figure 4, then the code fragment in Listing 4 can be used instead of the code fragment in Listing 3 to reconstruct coarsened superblocks instead of individual macroblocks, for an arbitrary block and picture size.

```
#pragma omp task input(*rc, *s, *ml, *mur) inout(*m)
void reconstruct_super_mb_task(MBRecContext *rc,
  Slice *s, Supermb *ml, Supermb *mur, Supermb *m){
  for (int k=0, i=mby; i< m->mby + sheight; i++, k++)
    for (int j= m->mbx -k ; j< m->mbx- k + swidth; j++){
      // if (i,j) is a valid macroblock
      if (i< rows && j>=0 && j<columns)
        reconstruct_mb(rc, s, i, j);
    }
}

#pragma omp task inout(*rc) input(*s,
  smbs[0;srows*scols]) output(*pic)
void reconstruct_task(MBRecContext *rc, Slice *s,
  Supermb *smbs, Picture *pic){
  init_ref_list(s);
  get_picture_buffer(rc, s);
  for(int i=0; i< srows; i++)
    for(int j=0; j< scols; j++){
      Supermb *m = &smbs[i*scols + j];
      Supermb *ml = m - ((j > 0) ? 1: 0);
      Supermb *mur = m - (((j < scols-1) && (i >0)) ?
        scols-1: 0);
      reconstruct_super_mb_task(rc, s, ml, mur, m);
    }
  Supermb *lastmb = &smbs[srows*scols -1];
#pragma omp taskwait on (*lastmb)
  release_ref(rc, s);
  *pic = s->pic;
}
```

Listing 4. Decoding tetris-shaped superblocks instead of single macroblocks.

The code in Listing 4 resembles the code in Listing 3. The main differences are the reduced loop boundaries smb_rows and smb_columns, calling the reconstruct_super_mb_task, and checking for valid macroblocks covered by the superblock before calling the inner reconstruct_mb function. This method is generally applicable to specify coarsened wavefront parallelism in OmpSs for arbitrary superblock sizes and picture dimensions with minimal runtime dependency checking overhead.

In the implementation so far, only the wavefront (WF) parallelism inside a frame is used. Wavefront parallelism, however, suffers from parallelism ramp-up and ramp-down, which reduces the average parallelism. By grouping the macroblocks the parallelism is further reduced and could limit performance scaling at higher core counts. The amount of parallelism can be increased by reconstructing multiple frames in parallel. A limitation is that the motion compensation kernel inside the reconstruction stage uses reconstructed pixel areas of previously decoded frames.

To reconstruct more than one frame in parallel additional dependencies must be expressed to superblocks/macroblocks of previous frames that produce the reference pixel areas. The motion vector pointing to the required pixel area is specific to each macroblock and is not know until they have been entropy decoded which poses a problem as the MBR tasks must be spawned before the ED tasks finish to exploit the pipeline parallelism. Some additional parallelism, however, can still be exploited by assuming the worst case motion vector size, which is 512 pixels in vertical direction. This assumption will allow the wavefront parallelism to overlap consecutive frames. The implementation of this technique will be referred to as the overlapping wavefront (OWF).

The additional dependency can be implemented by adding an extra `input` to the reconstruct_super_mb_task, which express a dependency to the macroblock producing the pixel area 512 pixels downwards of the previous frame. A consequence for the implementation is that the MBR tasks cannot be spawned in a nested task. This nesting would require that OmpSs performs dependence checking over different sub task graphs, which is not supported by OmpSs as this would break deterministic task dependency resolution. Instead the MBR tasks must be spawned in the main loop similar to the ED tasks.

Up to 14 cores, OWF with a block size of 12×12 has the highest performance, after which WF with a block size of 8×8 takes over. Due to the larger block size the additional parallelism of overlapping consecutive frames is evened out. The higher performance of OWF results from the more constant supply of free tasks. In the WF variant, the rampup and rampdown of the number of MBR tasks complicates locality aware scheduling. To outperform the WF at higher core counts, however, OWF requires faster dependency resolution with large task windows. Several works have investigated hardware support for OmpSs task dependency resolution [17], [18], [19].

IV. Other Usability Aspects of OmpSs

A. Compilation Toolchain

OmpSs requires the use of the Mercurium C/C++ compiler and the Nanos++ runtime library [20]. The Mercurium compiler interprets the OmpSs pragma annotated code and performs a source-to-source transformation to an intermediate code that contains the Nanos++ runtime library calls. The source-to-object compiling is then performed by a regular C/C++ compiler such as GCC. The Nanos++ runtime library implements the OmpSs runtime dependency checking and task scheduling.

To use OmpSs only a compiler switch is required. Because the Mercurium compiler accepts most of the GCC options, it is in almost all cases drop-in compatible with projects using the GCC compiler. For our benchmarks, this worked out-of-the-box for the C benchmarks. Some issues where found with the C++ support which currently is still experimental. Annotation of class member functions serializes the execution of these tasks and any C++11 support is missing.

Furthermore, error messages of the Mercurium compiler are often less descriptive than the ones produced by compilers such as Clang and even GCC. Also the Mercurium compiler sometimes does not behave well with incorrect input code, resulting in compiler crashes. These are all relatively small issues that could be worked around. Overall the compilation toolchain can be considered stable and usable, but still needs some polishing to be called production-ready.

B. Verification and Debugging

A common problem when developing parallel programs is verification and debugging. It is difficult to ensure that a parallel program is correct for all cases and debugging parallel programs, which have non-deterministic behavior, is non-trivial.

On the one hand, debugging OmpSs programs can be considered to be more difficult than debugging Pthreads programs for two reasons. First, the integration with traditional open source debugging tools like GDB [21] and Valgrind [22] is suboptimal. The source-to-source compilation nature makes stepping through code difficult because GDB shows the intermediate code. Also the Nanos++ runtime library triggers Valgrind errors due to switching to unregistered stacks. Finding real errors becomes difficult as they are flooded by errors generated by the runtime library calls.

Second, the OmpSs programming model gives the programmer an abstracted view of the underlying task execution system. In the case of program misbehavior, however, the raised abstraction level could complicate locating and eliminating bugs in cases where the programmer is not familiar with the underlying task execution model. This problem is currently more pressing due to a lack of documentation which describes the semantics and the interactions of OmpSs pragmas. This lack of documentation is partly due the fact that OmpSs is a novel programming model and still subject to extensions.

On the other hand, OmpSs can also be considered easier to debug due to tracing tools like Extrae [23] which is well integrated in the Nanos++ runtime. With Extrae, traces can be generated which show when and where the tasks are executed as well as dependencies with other tasks. Additionally, OmpSs ensures that the code without the pragmas is correct sequential C/C++ code which allows the programmer to first solve all the problems in the sequential code with conventional debugging tools. Furthermore, tools specific to help debugging OmpSs programs, like Starsscheck [24] and Ayudame/Temanejo [25] are currently being developed. Starsscheck checks if all the memory accesses made in the task function body are annotated correctly. Temanejo is a visual debugging interface to Ayudame, a StarSs task parallel debugger. Ayudame/Temanejo can simplify debugging considerably by visually rendering the task graph and allowing to manipulate and control task execution.

C. Source Code Size

The Pthreads benchmarks are in general larger than their sequential and OmpSs counterparts. This increase is due to the necessity of using multiple explicit threading and synchronization statements. OmpSs has a clear advantage since the only necessary additions to the code are **#pragma** statements to declare functions as tasks. Common parallel programming operations such as data replication and reductions have to be performed for both programming models. The code size difference between OmpSs benchmarks and their Pthreads counterparts, shown in Table III is mostly due to manual threading and synchronization.

Kernel	LOC abs.	LOC rel.	Workload / Application	LOC abs.	LOC rel.
c-ray	53	8%	rot-cc	142	12%
md5	21	2%	ray-rot	176	15%
rgbcmy	50	8%	streamcluster	142	14%
rotate	77	8%	bodytrack	241	3%
kmeans	84	13%	h264dec	600	3%

TABLE III
ADDITIONAL LINES OF CODE OF THE PTHREADS VARIANT COMPARED TO OMPSS VARIANT FOR EACH BENCHMARK.

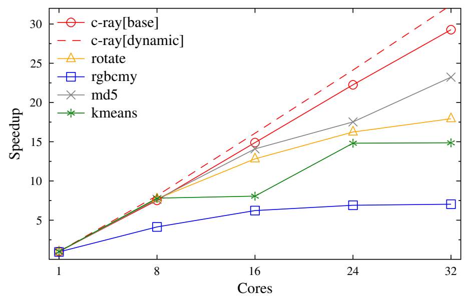
V. QUANTITATIVE EVALUATION

The performance of OmpSs is evaluated by comparing it against Pthreads using the ten benchmarks described in Section II-B. All the experiments are performed with the hardware and software setup described in Section II-C. Experiments have been performed using 1, 8, 16, 24 and 32 cores. For each core count the minimal number of sockets is used. In OmpSs this is done by default, while for Pthreads this is enforced using the `--physcpubind` option of the `numactl` tool. The number of cores used in OmpSs applications is controlled by an environmental variable.

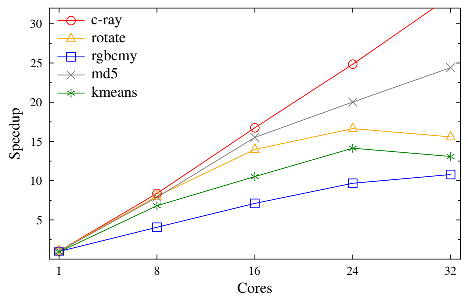
A. Overall Benchmark Scalability

Figures 5(a) to 5(f) show the speedup for the kernels, workloads, and applications for both OmpSs and Pthreads. The speedup is relative to the execution time of the *sequential* version of the benchmarks, which contains no parallel overhead. The figures show that not all benchmarks scale well. For 32 cores, the benchmarks parallelized with Pthreads achieve speedups in the range from 7.0× to 29.3×. OmpSs manages to achieve overall comparable speedups ranging from 9.1× to 33.5× with 32 cores.

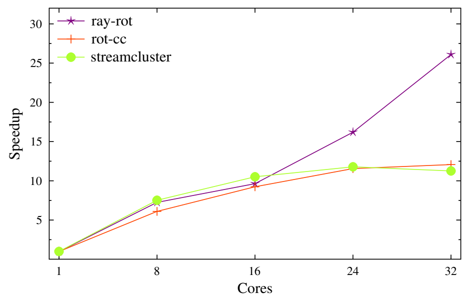
Both programming models show the highest scaling for the c-ray, md5sum and ray-rot benchmarks, due to their high computation to communication ratios. For the c-ray kernel there are two Pthreads versions. The base version divides the horizontal picture bands evenly over the cores, while in the dynamic version the lines of the picture are dynamically distributed using an atomic counter. The dynamic distribution



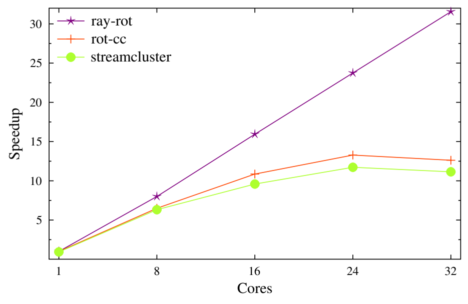
(a) K-type benchmarks, Pthreads



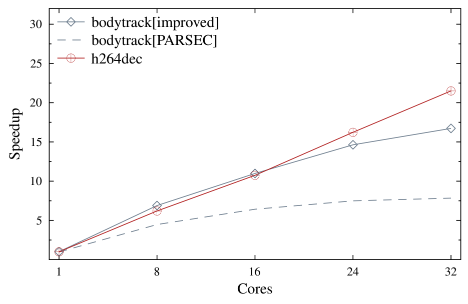
(b) K-type benchmarks, OmpSs



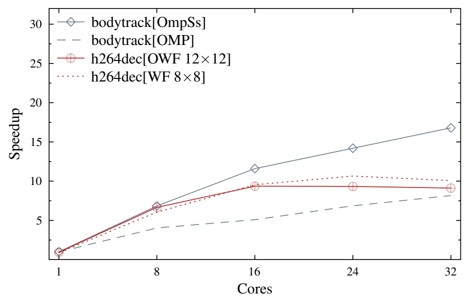
(c) W-type benchmarks, Pthreads



(d) W-type benchmarks, OmpSs



(e) A-type benchmarks, Pthreads



(f) A-type benchmarks, OmpSs

Fig. 5. Speedup results for all benchmarks. The results for the kernel, workload and application benchmarks are shown in different graphs.

improves performance, because not every line has the same amount of work. This dynamic distribution is performed automatically in OmpSs, and, therefore, the OmpSs version performs better out-of-the-box.

The lowest scaling occurs in the rgbcmy kernel with a maximum speedup of 7.0× and 10.8× for Pthreads and OmpSs, respectively. Neither programming model manages to achieve outstanding speedup for this kernel, because the benchmarks is memory-bound and the duration of the parallel phase is small. For similar reasons the scaling of the rotate, kmeans, rot-cc, and streamcluster benchmarks saturates before reaching 32 cores.

As described in Section II-B, an improved parallelization strategy is used in bodytrack compared to its original implementation of PARSEC 2.1. The figures show that the improved parallelization strategy is around two times more efficient, with 16.8× compared to 8.2× at 32 cores, indicating the importance of a balanced parallelization strategy. Furthermore, it quantifies the benefits of the additional expressiveness of OmpSs, since the improved parallelization strategy cannot be expressed using OpenMP.

B. Head-to-Head

In Table IV the speedup of the OmpSs variant over the Pthreads variant is shown for each benchmark and core count. For benchmarks with multiple version the version shown with a marker in Figures 5(a) to 5(f) are used. Overall five benchmarks are faster with OmpSs and four with Pthreads. The largest gains are observed with the c-ray, rgbcmy, and ray-rot benchmarks. The largest loss is observed with h264dec benchmark.

OmpSs performs better with the c-ray benchmark mainly due to the dynamic load balancing, because the Pthreads c-ray version with dynamic work distribution also achieves similar scaling. The OmpSs version even achieves superlinear speedups, which can be attributed to the automatic prefetching of task inputs performed by the runtime system.

In the rgbcmy benchmark multiple iterations are performed for run time stability, with a task/thread barrier separating each iteration. The absolute time for one iteration, however, is short

Benchmark	1	8	16	24	32	Mean
c-ray	1.03	1.11	1.12	1.11	1.14	1.10
rotate	1.06	1.04	1.09	1.02	0.86	1.01
rgbcmy	1.02	0.98	1.14	1.40	1.53	1.19
md5	1.00	1.02	1.10	1.14	1.05	1.06
kmeans	0.91	0.87	1.30	0.95	0.88	0.97
ray-rot	1.02	1.10	1.65	1.46	1.20	1.27
rot-cc	1.00	1.06	1.17	1.14	1.04	1.08
streamcluster	0.93	0.84	0.91	0.99	0.99	0.93
bodytrack	0.98	0.99	1.05	0.97	1.00	1.00
h264dec	0.94	1.07	0.87	0.57	0.42	0.73
Mean	0.99	1.00	1.12	1.05	0.97	1.02

TABLE IV
SPEEDUP FACTORS AND GEOMETRIC MEANS OF OMPSS OVER PTHREADS FOR EACH BENCHMARK AND CORE COUNT.

with less than 20ms on 16 cores. In this benchmark, the OmpSs variant is able to scale better at higher core counts because it employs a polling task barrier instead of the more expensive blocking thread barrier.

In the ray-rot benchmark the output of the c-ray kernel is the input of the rotate kernel. In the ray-rot benchmark OmpSs performs better than Pthreads, because the runtime task scheduler places depending tasks on the same core. Scheduling tasks that have an input output relation back-to-back on the same core improves cache locality. In the Pthreads implementation the two kernels are separated by a barrier. Interestingly, due to the locality advantage over Pthreads, the gains of the combined ray-rot workload exceed the gains of the individual c-ray and the rotate kernel.

The largest performance difference between Pthreads and OmpSs occurs in the h264dec benchmark. Figures 5(e) and 5(f) show that the performance results of the two are similar up to 8 cores, but are drifting further apart at higher core counts. In Section III-D we have shown that increasing the task granularity is necessary to improve the overall performance of OmpSs. By grouping the macroblock reconstruct tasks, however, the parallelism is limited, which in turn limits the performance at higher core counts. In the Pthreads version of h264dec the synchronization is highly optimized using a line decoding strategyand, therefore, grouping of tasks is not necessary. Also overlapped reconstruction of consecutive frames is less effective for OmpSs, due to increasing task overhead with larger task windows.

Over the entire benchmark suite, OmpSs performs 2% better compared to Pthreads. At 1 and 8 cores the performance is very close, while at 16 and 24 cores OmpSs is slightly faster. At 32 cores OmpSs is slightly slower mainly caused by the lower performance in the h264dec benchmark. Thus, we argue that performance wise OmpSs can compete with manual threaded solutions in the embedded and consumer benchmarks used in this paper.

To be a true alternative for manual threading, however, OmpSs processes must be able to dynamically share resources with other processes. Currently, OmpSs programs use a static number of cores controlled by an environmental variable. Furthermore, because the Nanos++ runtime implements core communication in a polling fashion for performance reasons, e.g. task barriers, all the used cores are always loaded fully even if there is insufficient work which reduces overall system responsiveness and power efficiency when too many cores are used.

VI. Related Work

A previous investigation of the programmability and performance of OmpSs was performed by Duran et al. [11]. In their work, they present the Barcelona OpenMP Task Suite (BOTS), consisting of several common HPC kernels, to evaluate the tasking capabilities of OpenMP. OpenMP tasks, however, lack the more intricate features of OmpSs such as automatic dependency resolution while these features are explicitly emphasized in our work. Additionally, Duran et al. only use HPC applications for their study and a comparison of OmpSs to other, more established programming models such as Pthreads is missing.

Podobas et al. [26] evaluated and compared three task- or function-based parallel programming models, OpenMP, Cilk++, and a novel approach, Wool. In their study, detailed performance analyses are performed, such as investigating the costs of creating, spawning, and joining tasks, leading to an overall performance analysis. The powerful OmpSs feature of delayed task execution, however, is not supported by any of the programming models used in their study and the applications used are, in contrast to our work, a small set of widely known HPC-like kernels such as FFT or Strassen. Information about the programmability of either of the programming models is not given.

An study conducted by Ravela [27] utilizes Intel TBB, Pthreads, OpenMP, and Cilk++ to inquire into the performance achieved with HPC benchmarks written using these models as well as the development time required to create the respective benchmark versions. While this approach attempts to objectively compare the development time using person-hours, it cannot deliver in-depth information on why development with one programming model might have taken longer than with another. In our work, we give key insights into several usability metrics of programming models such as the toolchain quality to make the development process as transparent as possible.

VII. Conclusions

In this paper, we have evaluated the applicability of OpenMP Superscalar to embedded and consumer applications. For this evaluation we have used ten benchmarks with different complexities and sizes, covering important embedded and consumer application domains. For comparability the benchmarks have been implemented in both Pthreads and OmpSs and exploit the same type of parallelism.

The case study of parallelizing H.264 decoding shows how a more complex parallelization strategy can be expressed in OmpSs, which captures the function-level as well as the nested data-level parallelism. This cannot be achieved in other task parallel programming models, such as the vanilla OpenMP and Cilk++, as they do not allow tasks to be spawned before they are ready to execute. The required dependency checking, however, introduces too much overhead for fine-grained tasks such as reconstructing a single macroblock to scale well. To achieve good performance these fine-grained tasks have to be coarsened to reduce the dependence checking overhead.

We also have commented on other usability aspects of OmpSs, namely the compilation toolchain, ways for verification and debugging, adaptability and portability, and source code size. To compile OmpSs programs a compiler change to the GCC compatible Mercurium source-to-source compiler is required. The toolchain works for most programs out-of-the-box, but is not yet production ready as it also serves as the research platform for OmpSs. A debugging feature OmpSs currently offers for debugging is automatic instrumentation to generate execution traces. Furthermore, a graphical task-based debugger is in development allowing manipulation of task execution. Finally, compared to thread-based programming models, OmpSs has a slight code size advantage.

From a performance perspective, over all ten benchmarks, OmpSs delivers performance comparable to Pthreads. Due to the built-in dynamic load balancing OmpSs is able to achieve better performance than statically load balanced Pthreads benchmarks while having a simpler implementation. The Pthread H.264 decoder, however, is up to two times faster at higher core counts because bundling tasks in OmpSs reduces the parallelism, which in turn limits the speedup at higher core counts. In the bodytrack benchmark, due to the higher expressiveness, OmpSs was able to use the improved parallelization strategy introduced in this paper. Using the improved parallelization strategy the performance of bodytrack increased by more than twofold compared to the PARSEC implementation.

Therefore, we argue that OmpSs is a potentially more viable programming model than other less expressive task-based parallel programming models for embedded and consumer applications. Like many other parallel programming models, however, a better integration with the operating system is required to increase resource awareness and composability.

VIII. Acknowledgements

This research has been supported by the European Community's Seventh Framework Programme [FP7/2007-2013] under the ENCORE Project (www.encore-project.eu), grant agreement n° 248647 [28]. Computational support has been provided by the Future SOC Lab of Hasso-Plattner-Institute Potsdam [29].

References

[1] L. Dagum and R. Menon, "OpenMP: A Proposed Industry Standard API for Shared Memory Programming," *IEEE Computing in Science and Engineering*, 1997.

[2] A. Gurtovoy and D. Abrahams, "The Boost C++ Metaprogramming Library," 2002.

[3] I. Molnar, "The Native POSIX Thread Library for Linux," RedHat Inc, Tech. Rep., 2003.

[4] Khronos Group, "OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems," 2009, http://www.khronos.org/opencl/.

[5] C. J. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. D. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang, "Intel's Array Building Blocks: A Retargetable, Dynamic Compiler and Embedded Language," in *Proc. 9th Annual IEEE/ACM Int. Symp. on Code Generation and Optimization*, 2011.

[6] NVIDIA, "CUDA: Compute Unified Device Architecture," 2007, http://developer.nvidia.com/object/gpucomputing.html.

[7] J. M. Perez, R. M. Badia, and J. Labarta, "A Dependency-Aware Task-Based Programming Environment for Multi-Core Architectures," in *Proc. IEEE Int. Conf. on Cluster Computing*, 2008.

[8] K. H. Randall, "Cilk: Efficient Multithreaded Computing," Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1998.

[9] J. Reinders, *Intel Threading Building Blocks*, 2007.

[10] J. M. Perez, R. M. Badia, and J. Labarta, "TECHNICAL REPORT 03/2007 A Flexible and Portable Programming Model for SMP and Multi-cores BSC-UPC," 2007.

[11] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguadé, "Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP," in *Proc. Int. Conf. on Parallel Processing*, 2009.

[12] M. Andersch, B. Juurlink, and C. C. Chi, "A Benchmark Suite for Evaluating Parallel Programming Models," in *Proc. Workshop on Parallel Systems and Algorithms*, 2011.

[13] C. Bienia and K. Li, "PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors," in *Proc. 5th Annual Workshop on Modeling, Benchmarking and Simulation*, 2009.

[14] "FFmpeg Libavcodec," http://ffmpeg.org.

[15] H. Vandierendonck, P. Pratikakis, and D. Nikolopoulos, "Parallel Programming of General-Purpose Programs Using Task-Based Programming Models," in *Proc. 3rd USENIX Workshop on Hot Topics in Parallelism*, 2011.

[16] A. Pop and A. Cohen, "A Stream-Computing Extension to OpenMP," in *Proc. 6th Int. Conf. on High Performance and Embedded Architectures and Compilers*, 2011.

[17] T. Dallou, B. Juurlink, and C. Meenderinck, "Improving the Scalability and Capabilities of the Nexus Hardware Task Management System," in *Proc. 1st Int. Workshop on Future Architectural Support for Parallel Programming*, 2011.

[18] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. Badia, E. Ayguadé, J. Labarta, and M. Valero, "Task Superscalar: An Out-of-Order Task Pipeline," in *Proc. 43rd Annual IEEE/ACM Int. Symp. on Microarchitecture*, 2010.

[19] C. Meenderinck and B. Juurlink, "A Case for Hardware Task Management Support for the StarSs Programming Model," in *Proc. 13th Euromicro Conf. on Digital System Design: Architectures, Methods and Tools*, 2010.

[20] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta, "Nanos Mercurium: a Research Compiler for OpenMP," in *Proc. 6th European Workshop on OpenMP*, 2004.

[21] GNU Project, "The GNU Project Debugger." [Online]. Available: http://www.gnu.org/software/gdb/

[22] N. Nethercote and J. Seward, "Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation," in *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2007.

[23] H. S. Gelabert and G. L. Sánchez, "Extrae," 2010. [Online]. Available: www.bsc.es/ssl/apps/performanceTools/files/ docs/extrae-2.1.1-userguide.pdf

[24] P. Carpenter, A. Ramirez, and E. Ayguadé, "Starsscheck: A Tool to Find Errors in Task-Based Parallel Programs," in *Euro-Par 2010 - Parallel Processing*, 2010.

[25] J. Gracia, "SMPSs Usage Basics," 2011. [Online]. Available: www.project-text .eu/sites/default/files/ics11_Basics.pdf

[26] A. Podobas, M. Brorsson, and K.-F. Faxen, "A Comparison of some recent Task-based Parallel Programming Models," in *Proc. 3rd Workshop on Programmability Issues for Multi-Core Computers*, 2010.

[27] S. C. Ravela, "Comparison of Shared memory based parallel programming models," Master's thesis, Biekinge Institute of Technology, 2010.

[28] Encore Project, "ENabling technologies for a future many-CORE." [Online]. Available: http://www.encore-project.eu/

[29] Hasso-Plattner-Institut Potsdam, "Future SOC Lab," http://www.hpi.uni-potsdam.de/forschung/future_soc_lab.html.