Expressing Inter-task Dependencies between Parallel Stencil Operations

Per Larsen, Sven Karlsson and Jan Madsen

DTU Informatics
Technical University of Denmark
{pl,ska,jan}@imm.dtu.dk

Abstract. Complex embedded systems are designed under tight constraints on response time, resource usage and cost. Design space exploration tools help designers map and schedule embedded software to complex architectures such as heterogeneous MPSoC's. Task graphs are coarse grained representations of parallel program behaviour which are used to evaluate the feasibility of a particular design. However, automatically extracting an accurate task graph from source code is challenging. This paper investigates how to describe data dependencies to aid tools based on program analysis in extracting task graphs from source code. We will examine a common parallel programming pattern – stencil operations – and show that even for such codes with a regular control flow, the precise dependencies between two stencil operations cannot always be determined at compile time.
We introduce a language construct which i) captures an upper bound on the number of dependencies between successive stencil operations and ii) instructs the compiler to generate code which ensures that the bound holds for each execution of the program.
The impact of our proposal is evaluated using a micro-benchmark and two soft real-time embedded image processing applications. The coding effort is low – at most one line of code per parallel loop was added. The performance impact is evaluated on a quad-core Linux workstation and we observe no statistically significant slowdown.

1 Introduction

Today's embedded systems are expected to process complex data in real time while being highly energy efficient and inexpensive to manufacture. The aforementioned goals are often in conflict so design decisions must be made. Design space exploration, DSE, tools can help designers make well-informed decisions by automatically finding and evaluating the trade-offs between the set of feasible designs [1,2,3].

To evaluate each design point, a coarse grained model of the application behavior must be extracted from its source code. Task graphs, which are directed acyclic graphs, are commonly used for this purpose. Each node in the graph represents sequential computation and each edge represents a precedence constraint

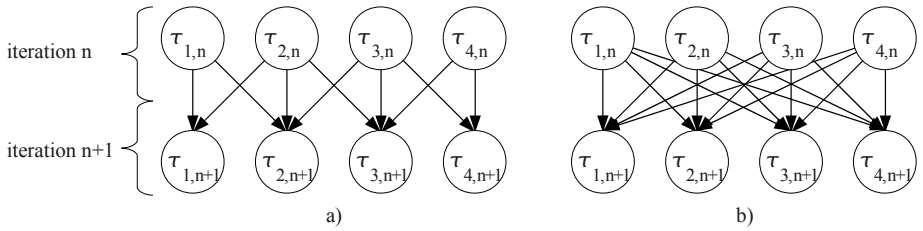

Fig. 1: a) Desired task graph for two iterations of heat diffusion example on four threads. b) Task graph fragment with over-approximation of inter-task dependencies.

between two tasks due to a data or control flow dependence in the source code [4].

Neither program analysis nor traces of program execution alone are sufficient to extract task graphs from source code. The former approach [5,6,7] computes a safe over-approximation of the inter-task dependencies for all program executions. Trace based approaches [8,9] can compute the precise dependencies between the tasks for a single program execution. However, trace based task graphs do not contain all possible dependencies since traces can only be generated for a tiny fraction of all possible program inputs.

In this paper, we show that it is possible to avoid some over-approximated edges and thus increase the accuracy of task graphs computed using program analysis by adding extra information to the source code through a programmer inserted directive.

Listing 1.1 contains a simple heat diffusion program parallelized with OpenMP [10] directives. It repeatedly applies a non-compact stencil which is five elements wide to a one dimensional array. The number of iterations in the inner loop and the number of threads are unknown. Stencil widths can also be unknown. The goal is to determine the dependencies between successive iterations of the *outer* loop.

Listing 1.1: Heat diffusion example. The code is adapted from Mattson et al. [11]

```
1  for(int k = 0; k < NSTEPS; ++k) {
2    /* instruct compiler to distribute iterations among available threads */
3    #pragma omp parallel for schedule(static)
4    for(int i = 2; i < NX-2; ++i) {
5      /* compute new value in 'ukp1' based on neighbouring values in 'uk' */
6      ukp1[i] = uk[i] + (dt/(dx*dx)) * (-1/12*uk[i-2]
7        +4/3*uk[i-1]-5/2*uk[i]+4/3*uk[i+1]-1/12*uk[i+2]);
8    }
9    /* swap pointers before executing next iteration */
10   tmp = ukp1; ukp1 = uk; uk = tmp;
11 }
```

Figure 1a shows a fragment of the task graph corresponding to two successive iterations of the outer loop *assuming* the number of iterations is at least twice the number of threads. Program analysis, however, cannot prove that each thread will receive more than one iteration of the inner loop – as thread and iteration counts are unavailable – and must therefore produce the task graph fragment shown in Fig. 1b.

Scope and Contributions We allow ourselves to assume that programs i) are parallelized with OpenMP ii) do not contain data races and iii) access arrays and dynamically allocated memory correctly.

This paper makes the following contributions:

- We introduce the `taskshare` directive (Section 3) which is necessary to derive task dependencies between parallel stencil operations when stencils wider than three elements are used.
- We also present the required runtime support (Section 4) so that object code generated by a compiler can check at runtime that the actual dependencies match the claims of the programmer inserted `taskshare` directives.
- The impact of the new directive is evaluated in terms of the resulting task graphs, coding effort and performance (Section 5). We use the heat diffusion example and two soft real-time embedded programs.

The ability of the directive to exclude dependencies otherwise reported by program analysis is directly proportional to the stencil width. When stencils spanning five elements are used, it enables a 40% reduction of dependencies between task pairs while stencils only three elements wide do not benefit from the directive. The dependencies excluded by the directive can allow DSE tools to find more feasible designs and prevent over-provisioning of resources. Secondly, less dependencies lowers the running times of task scheduling algorithms whose asymptotic complexity increase with the number of dependencies in the task graph [12,13,14,15].

In terms of coding effort, the impact is low – the `taskshare` directive adds at most one additional line of code to each loop performing a stencil operation. In terms of performance, benchmark measurements of three appliations show no statistically significant impact of runtime checks at the 95% confidence level.

2 Related Work

Vallerio et al. proposed an automated task graph extraction approach from unmodified C code [7]. Explicit parallelism is not supported and the extraction method relies on very conservative assumptions about the use of pointers and arrays which lead to task graphs containing dependencies which do not exist in the actual program. The challenge presented by pointer aliasing in context of task graph extraction was addressed in our previous work [16].

Adve and Sakellariou address task graph extraction in the context of high-performance computing [5]. They assume that a distributed memory programming model, MPI [17], is used exclusively for communication among tasks thereby making dependencies explicit. They also argue in favor of generating task graphs from OpenMP programs.

Ha has proposed the HoPES programming environment for development and mapping of embedded software to MPSoC's [18]. OpenMP is used to express data-parallelism in programs while task-parallelism is expressed in a synchronous data flow model. The difficulties of calculating dependencies among tasks representing parallel loops are not considered.

Liu and Dick present a tool which can generate communication graphs by tracing loads and stores during execution rather than compile time analysis [8]. Unlike task graphs, communication graphs cannot be parameterized or composed hierarchically to analyze application behavior across different program executions and execution platforms.

The hArtes project aims to develop an end-to-end development framework for real-time embedded systems [19]. The approach to task graph extraction is based on automatic parallelization of sequential C code. However, only a few types of code of can be parallelized automatically. Secondly, being able to derive a parallel program from a sequential program does not imply that an accurate task graph can be derived from the program. As this paper will explain, the number of dependencies between two parallel stencil operations depends on factors which are seldom available at compile time.

Schmitz et al. [1] determine inter-task dependencies by hand to schedule and map small applications to a smartphone. This approach is labour intensive and prone to errors.

3 Directive to Quantify Sharing among Tasks

We propose a new directive which can assert the maximal number of threads which will access the same array slice when executing a parallel loop.

The syntax of the directive in the informal notation of OpenMP [10] is: `#pragma depends` *[*`tskshare(`*expr*, *ts*`)` *...]* where *expr* must evaluate to either an array or a pointer to an array and *ts* must be the maximal number of threads concurrently accessing a slice of *expr*. The directive can only be used with an `omp for` directive and must appear directly before it. If several loops in a loop nest are parallelized with `omp for` then each should have its own `tskshare` directive. Finally, annotated loops must be coded such that the width of the stencil can be determined at compile time. This requirement is justified in Sect. 3.

The `tskshare` directive is useful when a sequence of stencil operations are encountered during task graph extraction in preparation for design space exploration. Program analysis can use the second argument of the tskshare directive to determine the number of dependencies from tasks belonging to stencil operation n to each task in stencil operation $n + 1$ instead of using a conservative approximation. In the heat diffusion example, the dependencies excluded are those which are in Fig. 1b but not Fig. 1a.

Correctness of the Directive At runtime, the iterations of a parallel loop are divided into a number of slices which in turn are mapped to threads. Consider a situation where the inner loop in the heat diffusion example contains 24 iterations which are divided into four slices of length six such that thread 0 executes iterations 0-5, thread 1 executes iterations 6-11, etc. Since the stencil width is five, thread 1 will read elements 4-13 which were written in the previous iteration of the outer loop – thread 0 wrote elements 4-5, thread 1 wrote 6-11

Listing 1.2: Innermost loop in the example with `taskshare` directive applied. The directive is underlined.

```
1  #pragma depends taskshare(uk,3)
2  #pragma omp for schedule(static)
3  for(int i = 2; i < NX-2; ++i) {
4    ukp1[i] = uk[i] + (dt/(dx*dx)) * (-1/12*uk[i-2]
5      +4/3*uk[i-1]-5/2*uk[i]+4/3*uk[i+1]-1/12*uk[i+2]);
6  }
```

and thread 2 wrote 12-13. The task executed by thread 1 therefore depends on three of its predecessor tasks. Generally the number of predecessors of a task equals the number of adjacent slices read plus one.

The number of adjacent array slices read by a task depends on the width of the stencil and the length of the slices – the wider the stencil and the shorter the length of each slice, the more adjacent slices are read. To check the correctness of the `taskshare` directive, the stencil width and minimal slice length must be computed to check that the number of adjacent slices read plus one is less than or equal to the value of `ts` argument in the directive. We note that the directive is not needed in case the stencil width is three because each task then reads at most three array slices regardless of their length.

Listing 1.2, shows the `taskshare` directive applied to the inner loop in the heat diffusion code. The `schedule` clause [10, p. 43] controls how iterations are divided into slices and mapped to threads. A certain slice length can be requested by the *chunk size* parameter. Since the schedule is `static` with no chunk size, each thread receives a slice with a length in proportion to the total number of iterations. The slice length is therefore greater than $\max(\lfloor \frac{n}{t} \rfloor, 1)$ for n iterations and t threads. Whenever $2n \geq t$, the length of each slice is at least 2 and since a five-point stencil can never access more than two elements of an adjacent slice, it accesses one adjacent slice on each side. Thus, in cases where the schedule is `static`, the stencil is five elements wide and the number of iterations is at least twice as great as the number of threads then the inserted directive correctly bounds the number of threads which concurrently access each slice in the loop.

Prerequisites for Runtime Checks The correctness of each `taskshare` annotation should be checked for correctness to guard against human error. Changes to i) the loop schedule or chunk size ii) the number of iterations iii) the stencil width or iiii) the number of threads can render the assertion captured by the directive invalid. None of these factors are required to be compile time constants but must be loop invariant [10] so checks must be performed at runtime.

To insert checks, the variables containing the values of the four factors must be identifiable by the compiler. The variables controlling loop schedule, iteration count and chunk size are trivially obtained. However, the width of the stencil is, in som cases, hard to identify at compile time.

Loops annotated with the `taskshare` directive must therefore adhere to three coding rules.

1. Do not convert multi-dimensional arrays to one-dimensional arrays.
2. Use array indexing expressions rather than pointer arithmetic.
3. Two `taskshare` clauses must not contain conflicting information on the number of threads accessing a possibly aliased array slice.

The first two rules ensure that program analysis can distinguish between accesses in different dimensions of a multi-dimensional array. The third criteria addresses pointer-aliasing. A `taskshare` clause specifies the number of threads that access an array slice. Aliasing may occur since the directive allows for the use of pointers to specify the array.

4 Runtime Checks

To measure the performance impact of the runtime checks, a library containing the necessary functionality was implemented. We also wrote a compiler plug-in which inserts runtime checks of `taskshare` directives during compilation. We used `llvm-gcc` version 2.5 with and `gcc` 4.2.1 which includes the `libgomp` OpenMP runtime. The exact nature of the runtime checks depends on how an OpenMP runtime implements the four different loop schedules. Our work is specific to `libgomp`.

If the schedule type of the parallel loop is either `static` or `dynamic` then the minimum slice length can be computed in constant time when the loop begins executing. However, if the schedule is `guided` then the length of each slice depends on all previously computed slices in the loop. The shortest slice is therefore found by comparing each slice as it is computed during the execution of the parallel loop – thus the overhead is in proportion to the number of slices. The `runtime` schedule is handled similarly to `guided`.

In case a runtime check detects a that a directive was not satisfied, it raises a runtime error. This implies that a task graph based on the assertions in `taskshare` directives does not match observable program behavior. It is left to the program to determine whether to continue execution or not. Runtime errors can be avoided either by modifying the `schedule` clause to lengthen the shortest slice, by modifying the directive, i.e. increasing the bound on threads concurrently accessing an array slice, or decreasing the stencil width if possible.

Inserting Runtime Checks during Compilation Figure 3 illustrates how the compiler transforms code annotated with an `omp for` directive.

The loop with a `static` schedule and unspecified chunk size in Fig. 3a is transformed into Fig. 3b whereas all other types of loops corresponding to Fig. 3c are transformed to Fig. 3d. In the first case, each thread receives its slice of the iteration space by means of a single call to `loop_start`. In the second case, each thread receives the initial slice from a call to `loop_start` and subsequent slices by calling `loop_next` at the end of the compiler inserted loop which encapsulates the original loop.

Our `llvm-gcc` plug-in adds an additional compilation step after the processing of OpenMP directives as illustrated in Fig. 2. In this step, the code generated

```
1  #pragma omp for schedule(static)
2  for(i=lb;i<ub;i+=incr)
3   /* <loop body> */
```

a) Parallel loop with `static` schedule and unspecified chunk size.

```
1  /* slice begin, slice end */
2  long sb, se;
3  if(loop_start(
4   lb,ub,incr,"static",0,&sb,&se) {
5   for(i=sb,i<se;i+=incr)
6    /* <loop body> */
7   }
8  }
9  loop_end();
```

b) Code generated for static schedule and default chunk size.

```
1  #pragma omp for schedule(...)
2  for(i=lb;i<ub;i+=incr)
3   /* <loop body> */
```

c) All other schedule choices.

```
1  /* slice begin, slice end */
2  long sb, se;
3  if(loop_start(
4   lb,ub,incr,ls,c,&sb,&se)) {
5   do {
6    for(i=sb,i<se;i+=incr)
7     /* <loop body> */
8   }
9   while(loop_next(&sb,&se));
10 }
11 loop_end();
```

d) Code generated for all other schedule choices.

Fig. 3: a-b) Translation of loop with `omp for` directive with `schedule(static)` clause into parallel loop. c-d) Translation of loop with `omp for` directive for all other `schedule` choices. Functions which may be replaced by wrapper functions containing runtime checks are underlined.

from the `omp for` directives is identified and runtime checks are inserted for loops annotated with the `taskshare` directive. When the minimal slice length can be calculated before entering the loop, i.e. when schedule type is `static` or `dynamic`, the call to `loop_start` is replaced by a call to `loop_start_wrapper`. The call performs a runtime check and then forwards the call to `loop_start`. Otherwise the call to `loop_next` is replaced with a call to `loop_next_wrapper`, which performs a runtime check on the length of the slice which was just processed and forwards the call to `loop_next`.

5 Experimental Results

We evaluate the required programming effort and how the runtime checks impact execution time of three parallel OpenMP benchmarks. We also evaluate the impact of the `taskshare` directive in terms of its ability to exclude dependencies between tasks which would otherwise have been computed by program analysis. This is only done on the first benchmark, however, as we have not been able to

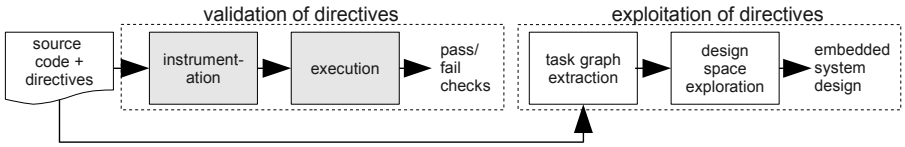

Fig. 2: Use of the `taskshare` directive in an embedded system design flow. Directives are validated by instrumenting the program and executing it to determine if all checks pass for all relevant program inputs. The information in the directives can then be exploited by a task graph extraction tool in preparation for the design space exploration step.

Name	Sequential time	Relative speed-up		Memory use
		2 threads	4 threads	
Heat diffusion	1.3781	1.9	3.1	2
Demosaicing	0.8980	1.9	3.5	221
Edge detection	0.5971	1.9	3.5	192

Table 1: Memory usage and speed-up of the benchmark applications run with `static` scheduling of loops. All times are in seconds and memory consumption is in megabytes.

obtain real embedded code using stencils wider than three elements. The embedded code can still be used to evaluate the runtime checks since the overhead of each check is independent of the stencil width.

The average execution time is calculated from thirty consecutive executions. We compare these averages with a two-sided, unpaired t-test using 95% confidence intervals. To quantify the utility of the directive, we study the relative change in the number of dependencies between a pair of tasks belonging to different stencil operations - e.g. tasks $\tau_{2,n}$ and $\tau_{2,n+1}$ in Fig. 1. Finally, the number of directives added in the source code is used to approximate the required programming effort.

5.1 Experimental Set-up

All experiments were performed on a workstation with a quad-core 2.66 GHz Intel Core i7 920 CPU and 3 GB DDR3 RAM. It had 256 KB L2 cache per core and 8 MB shared L3 cache. The operating system was 32-bit Ubuntu Linux 9.04 with kernel version 2.6.28. The measurements we will present were obtained using four threads but similar results were observed for experiments with one, two and eight threads.

Only the parts of the benchmark performing parallel work and which can contain our runtime checks are included in the execution time. The scalability of the parallelized parts of the benchmark applications is shown in Table 1.

We used the timing-facilities included in OpenMP which, on our platform, uses hardware cycle counters and obtains precision in the nanosecond range. To reduce the variability between executions we disabled dynamic power and frequency scaling, all hardware pre-fetching and simultaneous multi-threading (Hyper-Threading) via BIOS settings. The system ran in single user mode to reduce interference from background processes.

We have compared the quality of the code generated by `llvm-gcc` and `gcc`. This was done to ensure that the use of the LLVM optimizer did not do a poor job thus lowering the impact of our non-optimizable instrumentation. Our experiments show that that code generated by `llvm-gcc` is consistently faster than that of `gcc`. We used the optimization flag `-O2` which is supported by both compilers for all experiments.

5.2 Heat Diffusion

The heat diffusion code was used as a micro-benchmark. An `omp parallel` directive was put before the outer loop and an `omp for` replaced the `omp parallel for` to improve efficiency. It executes the same stencil operation iteratively for a number of time-steps so the inserted runtime checks are exercised repeatedly.

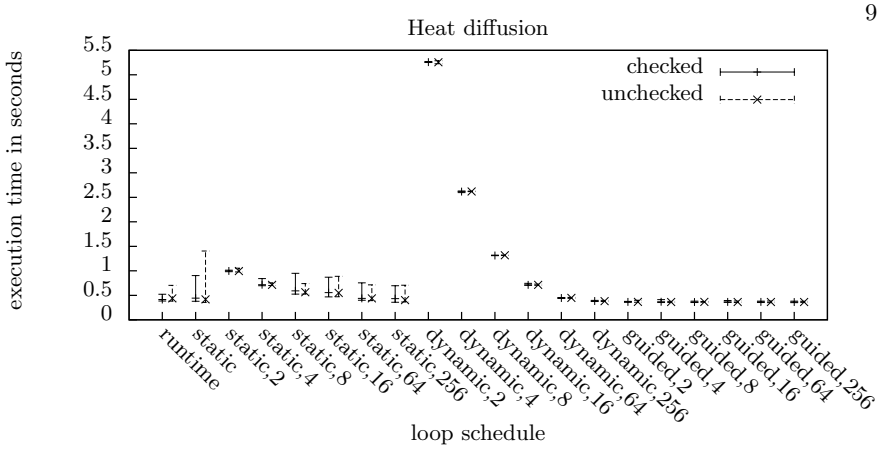

Fig. 4: Comparison of the maximal, minimal and average running times of *checked* and *unchecked* builds of the heat diffusion simulation with four threads and various selections of schedule and chunk size. The labels on the x-axis refers to the `schedule` clause.

The example uses a five-point stencil and thus the conservative estimate of inter-task dependencies among two executions of the stencil operation is five. Annotating the inner loop with a `taskshare` directive as shown in Listing 1.2 reduces that number to three between task pairs corresponding to a relative improvement of 40%.

The heat diffusion simulation was run for 2000 time steps with an array of 131072 `doubles`. The array sizes were set such that all data structures should fit in the CPU caches.

The running times of the heat diffusion simulation with and without runtime checks inserted are shown in Fig. 4. There were no statistically significant difference between the average running times of the binaries with and without runtime checks inserted.

The executions using the `dynamic` schedule and small values of the chunk size parameter have significantly higher execution times when compared to the other executions. This is because the `dynamic` schedule incurs a synchronization overhead each time a slice of iterations is mapped to a thread and the number of slices is in inverse proportion to the chunk size for the `dynamic` schedule.

Fig. 4 also shows a significantly higher worst-case execution time for uninstrumented builds for `runtime` and `static` schedules. Even when performing two or more warm-up runs before calculating average execution times, we saw that whichever build was executed first was also most likely to show a slightly higher worst-case execution time.

5.3 Demosaicing

An indispensable function in digital cameras is demosaicing which interpolates sensor data from a color filter mosaic. We used code developed for an embedded MPSoC [20]. It only uses stencils three elements wide which is too small to benefit from our `taskshare` directive but is nevertheless representative of a real

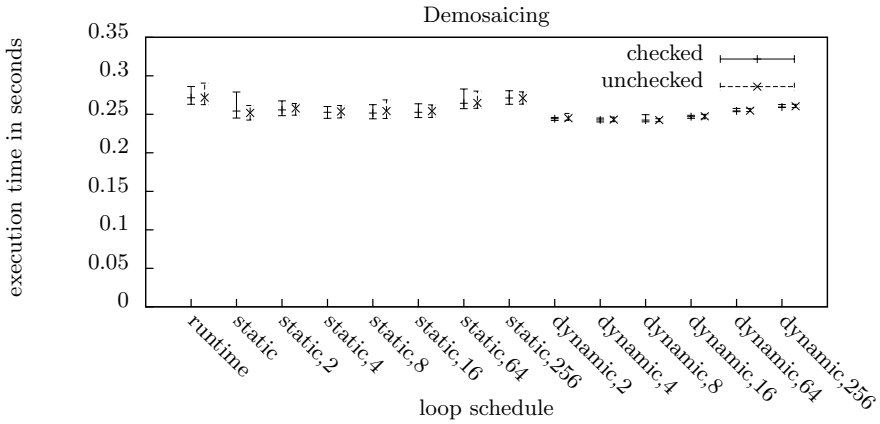

Fig. 5: Comparison of the maximal, minimal and average running times of *checked* and *unchecked* builds of the demosaicing program with four threads and various selections of schedule and chunk sizes.

world embedded application. Higher quality can be achieved with an algorithm using 5x5 stencils for interpolation [21]. A 21 mega-pixel image with a resolution of 5616x3744 pixels was used as input. The application contains three sequences of parallel stencil operations which were annotated.

No measurements were made with the `guided` schedule for this application since it triggers a known bug in the OpenMP runtime. All other results are shown in Fig. 5. There were no statistically significant differences between the average running times of the binaries with and without runtime checks inserted.

A slight increase in running time can be observed when increasing values of the chunk size parameter from 16 to 64 to 256 for the `static` schedule type. The effect also increases slightly when going from four to two threads. We are currently unable to give a reason for this but speculate that we are observing negative interference in the shared parts of the memory hierarchy, i.e., the L3 cache and memory controller.

5.4 Edge Detection

Detecting edges in an image is an important step in machine vision. We used the edge detection implementation which is part of the UTDSP benchmarking suite [22]. The input used consisted of 4096x4096 integer values.

The UTDSP edge detection code uses small stencils spanning only 3 elements which again does not require a `taskshare` directive. However, more advanced edge detection methods such as Canny's edge detection use dynamically computed Gaussian stencils whose width typically range from 5 to 19 elements [23]. Lacking a more advanced edge detection implementation, we annotated the UTDSP edge detection benchmark. It contains a method with a single parallel stencil operation which is called multiple times with Gaussian and Sobel filters as input so a single `taskshare` directive was added.

The results are shown in Fig. 6. There were no statistically significant differences in the average execution time with and without checks inserted.

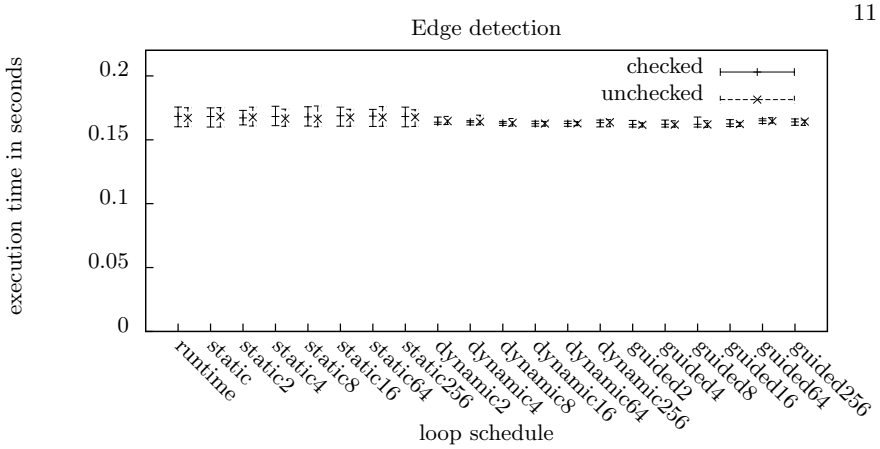

Fig. 6: Comparison of the maximal, minimal and average running times of *checked* and *unchecked* builds of the edge detection program with four threads and various schedules and chunk sizes.

6 Discussion and Conclusions

This paper presented a new directive to assist program analysis in computing inter-task dependencies. Specifically, it allows a reduction in the number of assumed dependencies in task graphs representing shared memory programs performing non-compact stencil operations. Such programs are found in embedded multimedia, machine vision, signal processing applications and in scientific computing.

Reducing the number of dependencies in task graphs i) can allow DSE tools to find more feasible designs and prevent over-provisioning of resources and ii) lower the running time of task scheduling algorithms whose asymptotic complexity increase with the number of dependencies.

When the stencil is compact, the inter-task dependencies can be correctly determined by program analysis. Whenever the stencil width is larger than three or unknown at compile time, the difference in the number of inter-task dependencies between the worst case and the common case grows with the number of threads.

The impact of the proposed directive was evaluated on the heat diffusion code, a single directive was added which results in a 40% decrease in the number of dependencies.

The embedded programs at our disposal used only 3x3 stencils and thus did not benefit from a `tasksharе` directive. We argue that this is due to the simplicity of the particular implementations and can point to algorithms which use wider stencils to produce a higher quality output.

In none of the cases were there a statistically significant increase in the average running times due to the insertion of runtime checks. However, the low-overhead runtime checking, presented in this paper rests on the fact that stencil operations access data in a highly regular manner – less predictable codes will require more elaborate runtime checks.

Acknowledgements This work was partially supported by HiPEAC[2] and Artist-Design, both European Union Networks of Excellence. The demosaicing example was provided by Polytechnique Montreal and STMicroelectronics Ottawa. The research also made use of the University of Toronto DSP Benchmark Suite, UTDSP.

References

1. Schmitz, M.T., Al-Hashimi, B.M., Eles, P.: System-Level Design Techniques for Energy-Efficient Embedded Systems. Kluwer Academic Publishers (2004)
2. Mahadevan, S., Virk, K., Madsen, J.: ARTS: A SystemC-based framework for multiprocessor systems-on-chip modelling. Des Autom Embed Syst **11**(4) (2007)
3. Gries, M.: Methods for evaluating and covering the design space during early design development. Integr. VLSI J. **38**(2) (2004)
4. Sinnen, O.: Task Scheduling for Parallel Systems. Wiley-Interscience (May 2007)
5. Adve, V.S., Sakellariou, R.: Compiler synthesis of task graphs for parallel program performance prediction. In: Proceedings of LCPC '00
6. Cosnard, M., Loi, M.: Automatic task graph generation techniques. In: Proceedings of HICSS '95
7. Vallerio, K.S., Jha, N.K.: Task graph extraction for embedded system synthesis. In: Proceedings of VLSID'03
8. Liu, A.H., Dick, R.P.: Automatic run-time extraction of communication graphs from multithreaded applications. In: Proceedings of CODES+ISSS '06, ACM
9. Ahmad, I., Kwok, Y.K., Wu, M.Y., Shu, W.: Casch: A tool for computer-aided scheduling. IEEE Concurrency **8**(4) (2000)
10. OpenMP Architecture Review Board: OpenMP application program interface, version 3.0. Technical report (2008)
11. Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming. Addison-Wesley (2004)
12. Sarkar, V.: Partitioning and Scheduling Parallel Programs for Multiprocessors. MIT Press (1989)
13. Yang, T., Gerasoulis, A.: DSC: Scheduling parallel tasks on an unbounded number of processors. IEEE Trans. Parallel Distrib. Syst. **5**(9) (1994)
14. Ahmad, I., Kwok, Y.K.: On parallelizing the multiprocessor scheduling problem. IEEE Trans. Parallel Distrib. Syst. **10**(4) (1999)
15. El-Rewini, H., Lewis, T.G.: Scheduling parallel program tasks onto arbitrary target machines. J. Parallel Distrib. Comput. **9**(2) (1990)
16. Larsen, P., Karlsson, S., Madsen, J.: Identifying inter-task communication in shared memory programming models. In: Proceedings of IWOMP '09
17. Snir, M., et al.: MPI – The Complete Reference, Vol. 1, The MPI Core, 2nd ed. The MIT Press, Cambridge, MA, USA (1998)
18. Ha, S.: Model-based programming environment of embedded software for MPSoC. In: Proceedings of ASP-DAC '07
19. Bertels, K., et al.: HARTES toolchain early evaluation: Profiling, compilation and HDL generation. In: Proceedings of FPL '07
20. Bouchebaba, Y., et al.: MPSoC memory optimization for digital camera applications. In: Proceedings of DSD '07
21. Li, J.S.J., Randhawa, S.: High order extrapolation using taylor series for color filter array demosaicing. In: ICIAR '05, Springer
22. Lee, C.G., et al.: UTDSP benchmark suite. http://www.eecg.toronto.edu/ corinna/DSP/infrastructure/UTDSP.html (1998) Date accessed: July 4th 2009.
23. Canny, J.: A computational approach to edge detection. IEEE Trans. Pattern Anal. Mach. Intell. **8**(6) (November 1986)