Stack Size Estimation on Machine-Independent Intermediate Code for OpenCL Kernels*

Stefano Cherubin
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano, I-20133 Milano, Italy
stefano.cherubin@polimi.it

Michele Scandale
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano, I-20133 Milano, Italy
michele.scandale@polimi.it

Giovanni Agosta
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano, I-20133 Milano, Italy
agosta@acm.org

This is a draft copy.

ABSTRACT

Stack size is a very important factor in the mapping decision when dealing with embedded heterogeneous architectures, where fast memory is a scarce resource. Trying to map a kernel onto a device with insufficient memory may lead to reduced performance or even failure to run the kernel. OpenCL kernels are often compiled just-in-time, starting from the source code or an intermediate machine-independent representation. Precise stack size information, however, is only available in machine-dependent code. We provide a method for computing the stack size with sufficient accuracy on machine-independent code, given knowledge of the target ABI and register file architecture. This method can be applied to make mapping decisions early, thus avoiding to compile multiple times the code for each possible accelerator in a complex embedded heterogeneous system.

CCS Concepts

•**Software and its engineering** → **Compilers;** •**Hardware** → *Emerging languages and compilers;*

Keywords

Stack size estimation; OpenCL

1. INTRODUCTION

The wide space margin provided by modern chip manufacturing techniques has given rise to a pervasive diffusion of a number of parallel computing architectures, up to the point where embedded systems are also characterized by multi- or many-core processors. Due to the need to minimize the energy budget, these multi-/many-core embedded processors are generally characterized by an heterogeneous architecture, where a small group of more powerful processors act as the "host", endowed with full capability to execute code, including the operating system, while a larger number of smaller processors act as one or more programmable accelerators.

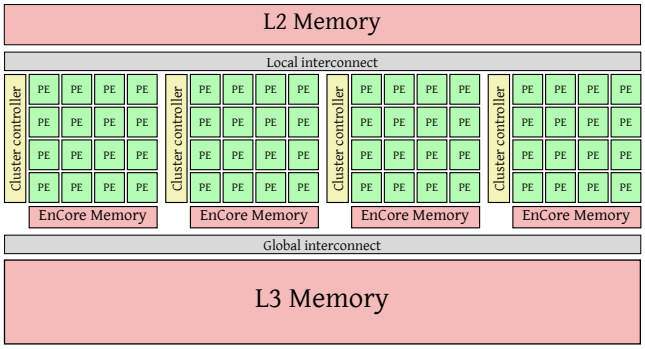

Figure 1: Architecture of STHORM/P2012, an example of Explicitly Managed Many Cores.

Architectures, such as STHORM/P2012 [12, 13] or PULP [7], exploit independent processing elements (PEs) grouped in clusters, with a small amount of tightly coupled dedicated memory bound to the cores, as shown in Figure 1. This class of architectures, sometimes called *Explicitly Managed Many-Cores*, combines several key characteristics, including scalability, power-efficiency, and the ability to leverage different types of parallelism — compared with traditional General Purpose GPUs, which are limited to massively parallel computation with little control flow divergence, Explicitly Managed Many-Cores support an independent control flow for each processing element.

To program these platforms, parallel programming models are typically used, such as the Open Computing Language (OpenCL) [9] or Open Multi-Processing (OpenMP) [3], both of which are supported, e.g., by the STHorm/P2012 software stack [6,11]. OpenCL describes the computation of a data-parallel program in terms of sets of work-items, called work-groups, which are mapped on the underlying architecture usually by the OpenCL runtime, while providing an architecture-agnostic structure to the programmer. This results in an ease of providing functional portability of parallel programs across different platforms. If the target device architecture does not provide enough hardware resources, the ability of OpenCL to provide functional code portability is disrupted – e.g., if the stack size for a kernel function grows too large, it will be impossible to run it on more limited devices.

In a simple system, if a kernel cannot be run on the accelerator due to memory constraints, it may be still executed on the host

*This work was supported in part by EU H2020-FETHPC program through the ANTAREX project under grant 671623.

ACM ISBN draft-copy.
DOI: draft-copy

as a fallback solution. However, in more complex system architectures, multiple heterogeneous accelerator devices may be available, or a single accelerator may be partitioned to allow concurrent execution of multiple OpenCL kernels. In this case, the mapping choices become more complex, and it is important for the compiler and resource management system to be able to estimate the memory requirements of a kernel, in order to perform the mapping task optimally.

In this work, we propose a technique to early estimate the stack size of functions before the machine-dependent intermediate code is generated. The proposed technique is based on the simulation of register allocation, performed before the machine-dependent code has been generated. We assess the accuracy of the prediction technique on a set of benchmarks. To do so, we implement the proposed technique, as well as a typical stack size analysis on machine-dependent code useful for comparison, in the industry standard LLVM compiler framework [10]. We provide an analysis of the accuracy of our estimation on the Scalable Heterogeneous Computing (SHOC) benchmark suite [4], a well known set of benchmarks designed for the OpenCL programming standard.

It is worth noting that the convergence between embedded and high performance computing systems, driven by the need to contain the power budget of high performance computing systems on one hand, and by the increased amount of functionalities requested from embedded and mobile systems, is leading to a *computing continuum* where established techniques from once separate domains are being adopted across larger portions of the continuum. Thus, clustered architectures not unlike the Explicitly Managed Many Cores are investigated as accelerators in the high-performance computing domain [5].

In this case, it may appear that memory size is less critical, since large amounts of memory are available in high-performance architectures. However, accelerators are in many cases deployed far from the larger memories, so that falling back to them imposes a significant memory penalty, which may undermine the capacity of the accelerator to actually provide a performance improvement over the host processors. Thus, the technique proposed in this paper may prove useful in a wider range of applications than its primary target.

Organization of the paper

The remainder of the paper is organized as follows. Section 2 provides a brief overview of related stack size analysis techniques. Section 3 describes our stack size estimation technique, while Section 4 reports experimental evidence to gauge its precision and usefulness. Finally, Section 5 draws our conclusions and points out some future investigations.

2. RELATED WORKS

The computation of stack size is useful for multiple scenarios. The most common is embedded software, which has to run on severely constrained hardware resources. In this scenario, stack overflow detection is the key issue, with the aim of increasing reliability. Tools such as Stackanalyzer [8] employ a combination of binary code analysis by abstract interpretation and user-provided information to compute precise and sound stack size information. Similar approaches are adopted in [1,2,15], all of which rely on the analysis of linked binaries.

The previously mentioned approaches differ from ours in goals and assumptions. In our case, the goal of the analysis is to assess as soon as possible whether a kernel can be mapped to a given accelerator or not, based on its memory requirements. Thus, we can accept some precision losses, but we want to gather as much information as possible before reaching the target-dependent backend stage of the compiler. Thus, analysing the binary is not a viable approach for our specific problem.

Other approaches rely on run-time analysis, i.e. stack size monitoring [14], once more to prevent stack overflow in more complex multiprocessing scenarios. Run-time analysis is neither necessary nor desirable in our scenario – on one hand, the stack size for accelerator kernel functions can be fully determined at compile time, so run-time analysis is not necessary, and on the other, the compiler needs this information before the code is deployed to the accelerator.

3. PROPOSED SOLUTION

In this section, we report the architecture of our proposed solution for the estimation of the stack size on machine-independent code. The approach can be divided in two main steps, the estimation of the stack size based on the knowledge of frame size estimates for all functions in the call graph of a kernel, and the estimation of the frame size of each function.

We generate two estimates – an *optimistic* one and a *pessimistic* one. The goal of the former is to minimize the absolute error of the estimation, whereas the goal of the latter is to provide a conservative estimate. Thus, the optimistic estimate is the one designed to provide a hint to the compiler to perform the mapping decisions, whereas the pessimistic estimate provides a more conservative solution which is useful for assessing the impact of source language and target machine features on the estimation algorithm.

Figure 2 shows the positioning in the LLVM compilation flow of the stack size estimation algorithm with respect to the stack size computation, which can be performed on machine-dependent code. Note that for OpenCL C kernels, where no recursion or function pointers are allowed, the stack size computation is precise, whereas for plain C code, it may not be so (in this case, our implementation detects and reports the issue).

3.1 Stack Size Estimation

Algorithm 3.1 reports the working of our solution for stack size estimation. The goal is to obtain, for each function f in the call graph CG a pair of stack size estimates, representing the *optimistic* and *pessimistic* estimate for the stack size respectively. Essentially, the algorithm works through the call graph by collecting first the leaf functions, i.e. those within which no further function calls are performed (line 4). For the leaf functions, the estimated stack size is merely the estimated frame size (lines 5-6). Then, the algorithm considers all functions within which calls are performed only to functions for which the stack size estimation has been computed (line 9). For these, the stack size is estimated as the sum of the estimated frame size and the largest stack size for any called function (lines 10-15).

Obviously, in case of recursive calls (either direct or indirect) to obtain an estimation one would need to estimate the recursion depth as well. In case of recursion, our algorithm reports a minimum stack size (assuming the recursive call is not performed) and signals the presence of recursion. However, in our main target programming model, OpenCL, recursion is not allowed in the kernel code, so there is no actual precision loss. For the same reason, we do not implement recursion depth estimation. For the sake of brevity, this special handling of recursion is not reported in Algorithm 3.1

3.2 Frame Size Estimation

Frame size estimation is performed through Algorithm 3.2. In this case, we need to take into account two sources of memory allocation in a function's frame: `alloca` LLVM IR instructions per-

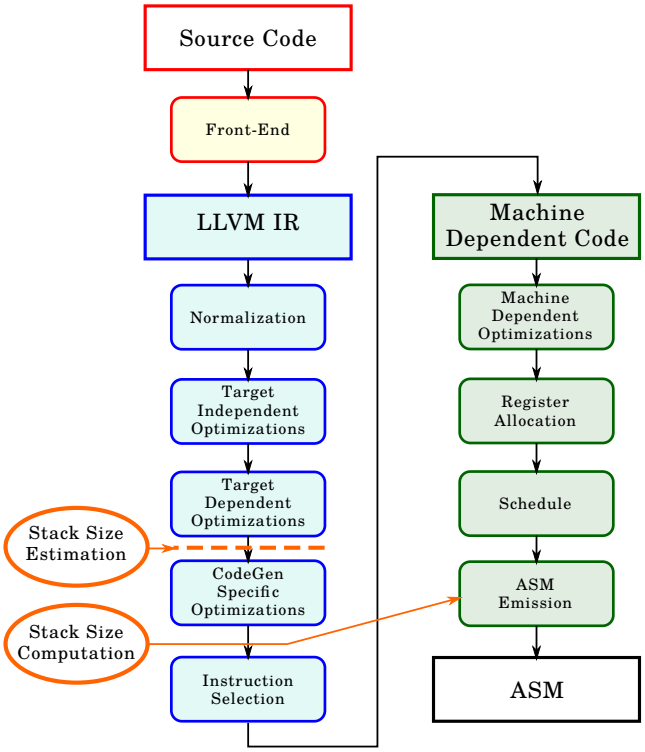

Figure 2: LLVM compilation flow, highlighting the position of the standard stack size computation pass (on machine-dependent code) and of our proposed pass. Pass borders are coloured as follows: in red, passes that operate on the source code; in blue, those working on the LLVM IR; in green, on machine-dependent code; in black, on assembly code. Each block is filled with a different colour depending on the stage of the compiler: yellow for the front-end; cyan for middle-end; and green for back-end.

Algorithm 3.1: Stack Size Estimation

```
1  Function StackSize (CG):
       Input: CG = (F, E) the Call Graph, where
              E ⊆ {(f, f')|f, f' ∈ F}
       Output: S, a set of triples
               (f, s_f^best, s_f^worst), f ∈ F ∧ s ∈ ℕ
2      begin
3          S ← ∅
4          F_ready ← {f|f ∈ F ∧ ∄f' ∈ F|(f, f') ∈ E}
5          foreach f ∈ F_ready do
6              S ← S ∪ {(f, FrameSize(f))}
7          end
8          while F ≠ F_ready do
9              F_new ← {f|f ∈ F ∧ ∄f' ∈ F \ F_ready|(f, f') ∈
                E}
10             foreach f ∈ F_new do
11                 s_f^best, s_f^worst ← FrameSize(f)
12                 s_f^best ← s_f^best +
                    max_{(f', s_f'^best, s_f'^worst) ∈ S ∧ (f,f') ∈ E} s_f'^best
13                 s_f^worst ← s_f^worst +
                    max_{(f', s_f'^best, s_f'^worst) ∈ S ∧ (f,f') ∈ E} s_f'^worst
14                 S ← S ∪ {(f, s_f^best, s_f^worst)}
15             end
16             F_ready ← F_ready ∪ F_new
17         end
18         return S
19     end
20 end
```

formed to explicitly allocate data in the stack and spilled variables from register allocation. For the former, it is sufficient to compute the array size (lines 13-16) and add it to the stack size (lines 25-26). For the latter, we perform *Liveness Analysis* beforehand, so as to know which values are alive at every point in the function (the outcomes of the analysis are used at line 17). Then, we simulate the *Register Allocation* algorithm, to detect which values are stored in registers, and which are spilled to memory (line 18). In our *optimistic* estimate, we consider all values held in registers to not need allocation on the stack, whereas in the *pessimistic*, we assume all values need to be saved on the stack as long as they are alive (lines 19-20).

Our register allocation simulation algorithm assumes the size and type of available registers on the specific target architecture to be known. This assumption impacts only on the *optimistic* estimation and does not affect at all the *pessimistic* estimation. In order to perform the simulation, live values are grouped in five sets, sorted by element size: (1) scalar variables from wider to smaller, (2) vectors from wider to smaller, (3) struct from smaller to wider, (4) arrays from smaller to wider and (5) other values from wider to smaller. Allocation starts with set (1) and proceeds with sets (5), (2), (3) and (4). For each value we attempt to find the smaller available register that is wide enough to contain it. Whenever it is possible, float and vector values are allocated respectively to float and vector registers. Otherwise, we use general purpose or integer registers.

In case of values wider than the size of the wider available register, we attempt to allocate these values on multiple smaller homogeneous registers. Every time we cannot allocate a value in the simulated register file, we increase the size of the frame size, including the alignment (lines 3-4). In the optimistic estimate, we assume no padding is needed for alignment, whereas in the pessimistic scenario we consider a fixed alignment, based on the target architecture.

3.3 Sources of Inaccuracy

While we simulate closely the workings of the Register Allocation algorithm, and identify accurately the values generated and used by each instruction in the machine-independent intermediate representation, there are still several sources of inaccuracy in the estimate.

First, our analysis does not take into account machine-dependent optimization passes. These may significantly reduce the amount of memory used by folding temporaries generated and used by sequences of instructions that can be mapped to a single machine instruction. This effect has a stronger impact on CISC instruction sets, as well on instruction set extensions that deal with specialized data types, such as vector extensions.

To mitigate this issues, it is possible to take into account machine-dependent effects, without performing all the code generation steps. For example, in machines where the conditions are stored in the processor status word rather than in explicit registers, boolean variables present in the intermediate representation are usually folded. Heuristically, it is possible to assume they require no storage in memory, even though the basic analysis would tell us otherwise.

4. EXPERIMENTAL EVALUATION

In this section, we report the results of the proposed stack size estimation method. We employ two suites of benchmarks: (1) a set of internally developed digital signal processing (discrete cosine

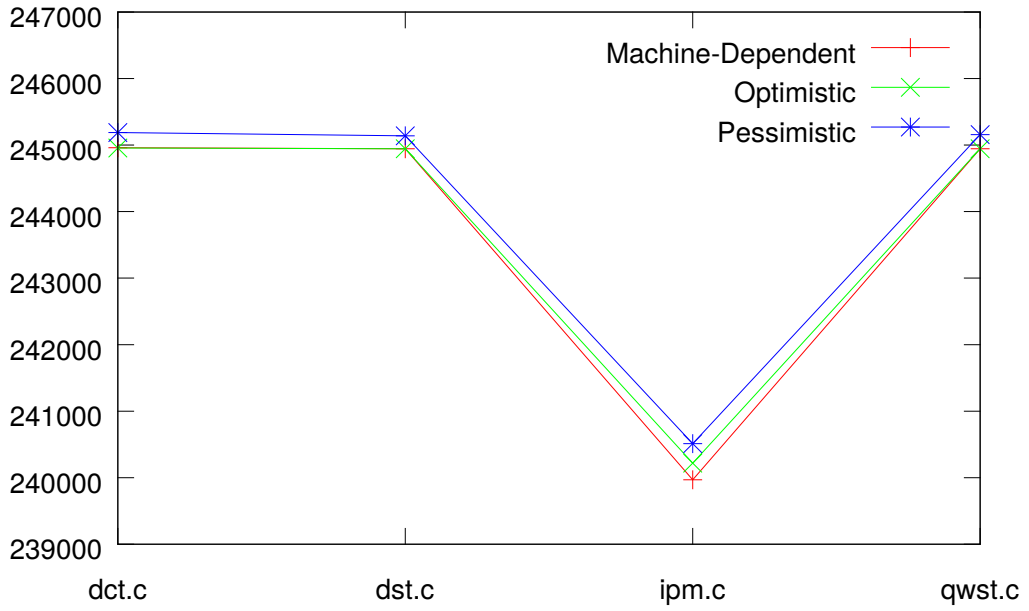

Figure 3: Digital signal processing and Image Processing benchmarks. The three lines report respectively the stack size computed after instruction selection on the machine-dependent code (*Machine-Dependent*), and the stack size estimate computed using our *Pessimistic* and *Optimistic* methods.

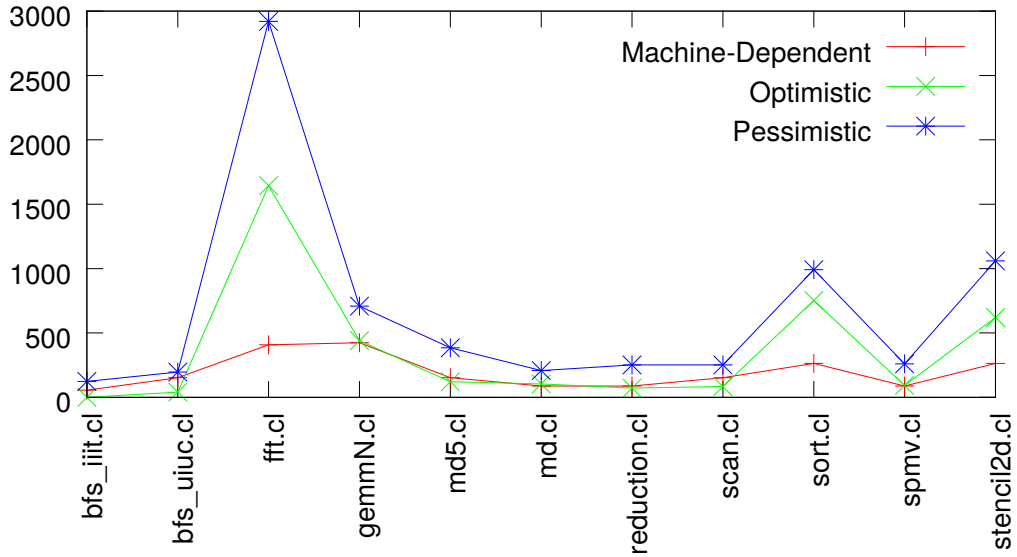

Figure 4: SHOC benchmarks, level one suite. The three lines report respectively the stack size computed after instruction selection on the machine-dependent code (*Machine-Dependent*), and the stack size estimate computed using our *Pessimistic* and *Optimistic* methods.

transform, discrete sine transform, quarter-wave sine transform) and image processing morphological transformation (dilation, erosion, laplacian filter), written in C language without involving recursion; (2) level one benchmarks of the Scalable Heterogeneous Computing (SHOC) benchmark suite [4].

The code for each benchmark is compiled for the *x86_64* instruction set architecture using clang 3.8. Due to the large instruction set, *x86_64* poses a greater challenge to the estimation algorithm with respect to simpler instruction sets – preliminary results on the MIPS architecture are in line with this assumption. The stack size is estimated before reaching machine-dependent code, as well as after the code generation phase, when the machine-dependent code

is available. The latter stack size estimate is employed to gauge the accuracy of the machine-independent code estimation. This is particularly important because one of the main source of inaccuracy for the estimation algorithm is the application of target-dependent optimization passes, which are not accounted for in the early estimation.

Figure 3 reports the analysis of the digital signal processing and image processing benchmarks written in the C language. The stack size is estimated with an error of less than 0.2% using the optimistic heuristic. It is underestimated in the case of the discrete sine transform by 4 Bytes, overestimated by 252 Bytes for the image processing kernels, and correctly estimated for the remaining benchmarks. The pessimistic computation always overestimates the stack size, by less than 0.3% (with a maximum error of 544 Bytes).

In the case of the SHOC benchmark suite, reported in Figure 4, the presence of OpenCL-specific constructs, in particular vector data types, causes some loss in precision. However, the *pessimistic* analysis is still able to provide a conservative estimate of the stack size, while the *optimistic* one provides usually an accurate estimate. It is worth noting that in OpenCL benchmarks the stack size is much smaller, so the relative impact of these errors is larger than in the C benchmarks. The optimistic heuristic overestimates the stack by 1236 Bytes in the case of the `fft` benchmark, but the average estimation error on all other SHOC benchmarks is lower than 120 Bytes. It is worth noting that `fft` is, among the benchmarks, the one that makes the heaviest use of vector data types, which are a major source of inaccuracy in the current algorithm.

5. CONCLUSIONS & FUTURE DEVELOPMENTS

We have presented a method to compute an approximation of the stack size of OpenCL kernels (and, more generally, C functions that need to be offloaded to heterogeneous accelerators). The goal of the stack size approximation is to enable the compiler to make a mapping decision (whether to run the kernel on an accelerator or not, and on which accelerator if more than one is available) before generating the machine-dependent code, so as to reduce the overhead of just-in-time compilation of OpenCL kernels.

The computed estimation can still be improved, by taking into account the impact of target-dependent optimization and instruction selection on the amount of memory used. To this end, we plan to characterize each target architecture through learning techniques, by associating a likelihood that a variable is folded by later optimization to each pair of instructions connected by def-use relations. We espect this extension to significantly reduce the estimation error of our optimistic heuristic.

6. REFERENCES

[1] Gogul Balakrishnan and Thomas Reps. WYSINWYX: What You See is Not What You eXecute. *ACM Trans. Program. Lang. Syst.*, 32(6):23:1–23:84, August 2010.

[2] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. End-to-end Verification of Stack-space Bounds for C Programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 270–281, New York, NY, USA, 2014. ACM.

[3] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008.

[4] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, pages 63–74, New York, NY, USA, 2010. ACM.

[5] José Flich, Giovanni Agosta, Philipp Ampletzer, David Atienza Alonso, Alessandro Cilardo, William Fornaciari, Mario Kovaĉ, Fabrice Roudet, and Davide Zoni. The MANGO FET-HPC Project: An Overview. In *Proceedings of the 18th IEEE Conference on Computational Science and Engineering, Special Session on FET-HPC and Exascale Recently EU-Funded Projects*. IEEE Computer Society, October 2015.

[6] Davide Gadioli, Simone Libutti, Giuseppe Massari, Edoardo Paone, Michele Scandale, Patrick Bellasi, Gianluca Palermo, Vittorio Zaccaria, Giovanni Agosta, William Fornaciari, and Cristina Silvano. OpenCL Application Auto-tuning and Run-Time Resource Management for Multi-core Platforms. In *Parallel and Distributed Processing with Applications (ISPA), 2014 IEEE International Symposium on*, pages 127–133, Aug 2014.

[7] Michael Gautschi, Andreas Traber, Antonio Pullini, Luca Benini, Michele Scandale, Alessandro Di Federico, Michele Beretta, and Giovanni Agosta. Tailoring instruction-set extensions for an ultra-low power tightly-coupled cluster of OpenRISC cores. In *Very Large Scale Integration (VLSI-SoC), 2015 IFIP/IEEE International Conference on*, pages 25–30, Oct 2015.

[8] Daniel Kästner and Christian Ferdinand. Proving the absence of stack overflows. In Andrea Bondavalli and Felicita Di Giandomenico, editors, *Computer Safety, Reliability, and Security*, volume 8666 of *Lecture Notes in Computer Science*, pages 202–213. Springer International Publishing, 2014.

[9] Khronos Group. The Open Standard for Parallel Programming of Heterogeneous Systems, (last accessed Aug. 2014).

[10] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[11] Andrea Marongiu, Alessandro Capotondi, Giuseppe Tagliavini, and Luca Benini. Improving the programmability of STHORM-based heterogeneous systems with offload-enabled OpenMP. In *Proceedings of the First International Workshop on Many-core Embedded Systems*, pages 1–8. ACM, 2013.

[12] Diego Melpignano, Luca Benini, Eric Flamand, Bruno Jego, Thierry Lepley, Germain Haugou, Fabien Clermidy, and Denis Dutoit. Platform 2012, a Many-core Computing Accelerator for Embedded SoCs: Performance Evaluation of Visual Analytics Applications. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1137–1142, New York, NY, USA, 2012. ACM.

[13] Julien Mottin, Mickael Cartron, and Giulio Urlini. The STHORM Platform. In *Smart Multicore Embedded Systems*, pages 35–43. Springer, 2014.

[14] SungHo Park, DongKyu Lee, and SoonJu Kang. Compiler-Assisted Maximum Stack Usage Measurement Technique for Efficient Multi-threading in Memory-Limited Embedded Systems. In Roger Lee, editor,

Computers,Networks, Systems, and Industrial Engineering 2011, volume 365 of *Studies in Computational Intelligence*, pages 113–129. Springer Berlin Heidelberg, 2011.

[15] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(4):751–778, 2005.

Algorithm 3.2: Frame Size Estimation

Function `Size` *(i)*:
  Input: i an instruction that defines a new value
  Data: $align_{min}$, $align_{max}$: minimum and maximum alignment padding
  Output: Best and worst case size for the value defined by i
  begin
    $size_{min} \leftarrow$ `sizeof(type(`i`))` $+ align_{min}$
    $size_{max} \leftarrow$ `sizeof(type(`i`))` $+ align_{max}$
    return $size_{min}, size_{max}$
  end
end

Function `FrameSize` *(f)*:
  Input: f a function in the Call Graph
  Data: `opcode` (i): retrieves the opcode for instruction i; `alloca`: the opcode for the LLVM IR instruction that performs explicit allocation in the stack frame
  Output: Best and worst case frame size for function f
  begin
    $s^{best}_{alloca} \leftarrow 0$
    $s^{worst}_{alloca} \leftarrow 0$
    foreach $bb \in f$ **do**
      foreach $i \in bb$ **do**
        if `opcode`$(i) =$ `alloca` **then**
          $s^{best}_{alloca} \leftarrow s^{best}_{alloca} +$ `Size`$^{best}(i)$
          $s^{worst}_{alloca} \leftarrow s^{worst}_{alloca} +$ `Size`$^{worst}(i)$
        end
        $live_i \leftarrow$ `LivenessAnalysis(`i`)`
        $inRegs_i \leftarrow$ `RegisterAllocation(`i`)`
        $s^{best}_i \leftarrow \sum_{i' \in live_i}$ `Size`$^{best}(i') - \sum_{i' \in inRegs_i}$ `Size`$^{best}(i')$
        $s^{worst}_i \leftarrow \sum_{i' \in live_i}$ `Size`$^{worst}(i')$
      end
      $s^{best}_b b \leftarrow \max_{i \in bb} s^{best}_i$
      $s^{worst}_b b \leftarrow \max_{i \in bb} s^{worst}_i$
    end
    $s^{best}_f \leftarrow s^{best}_{alloca} + \max_{bb \in f} s^{best}_{bb}$
    $s^{worst}_f \leftarrow s^{worst}_{alloca} + \max_{bb \in f} s^{worst}_{bb}$
    return s^{best}_f, s^{worst}_f
  end
end