SEVENTH FRAMEWORK PROGRAMME

FP7-215216

Architecture Paradigms and Programming Languages for Efficient programming of multiple COREs

Specific Targeted Research Project (STReP) THEME ICT-1-3.4

Final report of benchmark evaluations in different programming paradigms

Deliverable D2.3, Issue 0.1

Workpackage WP2

Author(s):	D. Rolls, C. Joslin, S.-B. Scholz, C. Jesshope, R. Poss		
Reviewer(s):	C.Jesshope, S.-B. Scholz		
WP/Task No.:	WP2	**Number of pages:**	64
Issue date:	2011-07-31	**Dissemination level:**	Public

Purpose: This deliverable evaluates the programming paradigms in Apple-CORE and their effectiveness on the Microgrid in comparison with legacy hardware in the domains of high-performance computation, embedded systems and mainstream applications.
Results: A range of critical benchmarking kernels and applications have been shown to be auto-parallelisable with the SAC2C compiler and the data parallel code that the SAC2C compiler produces has been shown to perform and scale well when simulated on the Microgrid architecture. Moreover functional concurrency is available for additional acceleration even when data parallelism also occurs. Legacy C code can also benefit from the APPLE-CORE many-core technology and in many cases can scale as well as non legacy code. The non-legacy code can however parallelise much code that is known to be extremely difficult to tackle in legacy code.
Conclusion: The Microgrid architecture shows great potential for applications specialised for high performance computing, for embedded computing and for general purpose computing. High-level languages like SAC expose data parallelism on a large scale and tool-chains can make use of the architecture to produce efficient parallel code utilising hardware threads to compute this code. In addition vast amounts of parallelism can be generated and exploited in legacy code through loop analysis techniques. We have demonstrated that hardware controlled threads is a good idea but also that the software must no be completely left out from thread distribution decisions.

Approved by the project coordinator: Yes **Date of delivery to the EC:** 2011-07-31

Document history

When	Who	Comments
2010-11-01	Daniel Rolls	Initial version
2011-01-25	Daniel Rolls	Graph display commands and first graphs
2011-01-26	Carl Joslin	Added more Livermore Loop graphs
2011-03-23	Daniel Rolls	Replaced gappy graph with those run on the latest compilers
2011-03-25	Daniel Rolls	Initial Matmul Section
2011-05-12	Daniel Rolls	Added graphs to the Matmul section
2011-05-22	Daniel Rolls	Added Livermore Loop and N-Body graphs
2011-05-24	Daniel Rolls	Fixed up some of the analysis in the Matrix multiplication section
2011-06-21	Daniel Rolls	Added SL Livermore Loop 9 graph
2011-07-11	Daniel Rolls	Updated the Quicksort graphs and some Livermore Loop graphs
2011-07-13	Daniel Rolls	First draft of Conclusion
2011-07-19	Sven-Bodo Scholz	Improved Livermore Loop results
2011-07-24	Chris Jesshope	Added dissemination activities
2011-07-26	Daniel Rolls	General clean-up
2011-07-28	Raphaël Poss	Simulated architectures section
2011-07-29	Sven-Bodo Scholz	Some final brushing

SEVENTH FRAMEWORK PROGRAMME

Project co-funded by the European Commission within the 7^{th} Framework Programme (2007-11).

Table of Contents

Section 1 - Introduction

This report evaluates the programming paradigms in Apple-Core and their effectiveness on the Microgrid. It follows on from Apple-Core Deliverable 2.2 where application benchmarks were selected. All partners on the project were encouraged to select from a range of the application benchmarks representing high performance computation systems, embedded systems and mainstream applications. Partners have also worked together to unify experiments with some of these benchmarks and allow comparisons to be between the various tool-chain paths within the Apple-Core project.

This document is split into a number of benchmark areas and within each section experimental results are presented with some analysis. Many of the results were produced using the Unibench system described in Apple-Core Deliverable 2.1a. All graphs produced using this system have a hyperlink in their captions to the graph on the Unibench system. From here the results in the Unibench systems and all information used to recreate the experiments that produced these figures is available. The majority or results are simulated which means that given the same, source code, compiler versions, runtime inputs and compiler parameters the experiments can be repeated to verify the exact same results. Each section that contains results from Unibench also has a convenient hyperlink to the Unibench system for quick access to all the source code used in the experiments. Source code exists on the system for experiments run for Apple-Core in C, SL and SaC.

Whereas this report summarises the benchmarking and evaluation effort in Apple-Core, the project was successful in breaking new grounds with both, dissemination and evaluation. The former through automated publications of results on a public forum and the latter with automated running of the experiments that produce these results.

Section 2 outlines the dissemination avenues beyond this report that present the materials of the Apple-Core project in a detail far beyond what can be provided here. Note that for the simulated results on the Microgrid many of the results were run on the same simulated architecture. For this reason Section 3 briefly outlines this architecture. The report then moves on to looking at benchmarking experiments in Section 4 before some concluding remarks in Section 5.

Section 2 - Dissemination Activities

This section discuses dissemination efforts realated to benchmark performance evaluation within Apple-Core.

2.1 Unibench

In addition to this report most results in Apple-CORE are available via the Unibench [RHJS09] web service at `unibench.apple-core.info`. Whilst this service was originally intended as a tool to facilitate the measuring process within a highly complex evolving tool-chain in later stages of Apple-CORE it became useful as a means to dissemination of results. For all experiments Unibench provides public access to source code, runtime inputs, compiler flags and measurement scripts used. It also provides version strings identifying the tool chain builds and machines used in the process. Effectively the information provided on Unibench allows an experiment to be repeated and on Apple-CORE where experiments are mainly simulated this simulation should be predictably repeatable in a way that real hardware isn't. The information provided on Unibench is far more comprehensive than what could possible be provided in a report and far more accessible and so we hope that unibench compliments and supplements the analysis in this report.

2.2 Website

The Apple-Core website is the main avenue for dissemination of information from the project. It serves as a reference to the project providing background information and an elevator pitch and as a public archive of all public deliverables. References to all publications funded by the Apple-Core project are listed there and events organised within the consortium are promoted there. The website is located at `www.apple-core.info`.

2.3 Publications

Publications funded by Apple-Core will undoubtedly be excepted into proceedings after the publication of this report. An up-to-date comprehensive list of publications is maintained at `http://www.apple-core.info/publications/`.

2.4 Exploitation

The work in the Apple-Core project is enabling and the main results planned are a complete set of tools that enable the Apple-Core approach to many-core processors to be evaluated across a wide range of applications and processor configurations, using several different programming paradigms, from assembly code, through system-level languages to high-level and high productivity languages. Moreover all of these tools will be made available in the public domain for further investigation. Thus, at this stage, we should measure the initial success of the exploitation in terms the uptake of these tools and their adoption, modification, evaluation in institutions outside of the project. Note that we do not expect the core results of the project to be immediately adopted by the mainstream processor industry. However with sufficient engagement over the next few year this could be possible in the 5-10 year timeframe, see *Exploitation Roadmap for the Apple-CORE Results* in the Apple-Core final report for more details.

Because of the pre-competetive nature of the outputs of this project our dissemination activities have been broad based and include:

Mainstream processor manufacturers. Here we have been in discussions with two of the major commodity processor manufacturers (Intel and Oracle - formerly Sun Microsystems) as well as a major software company. Due to the nature of the processor design business, it is unlikely that the results of the project will be exploited directly by any of these companies. However all are interested in the concurrency abstractions and programming models we have been developing. This interest has also been reified in terms of collaboration in a nationally funded project (Intel is an unfunded partner in the NWO MODERN project, which is looking at power issues in SVP systems) and also donation of hardware. In the last year we have received an Intel SCC system at UvA, access to an Intel SCC system at UH and also loan of a Niagara T3 system, with the expectation we will get a loan of the T4 Niagara prior to release. All for evaluation in relation to implementing the SVP system-level model in software and exploitation of the tool chain above this. Other tangible indicators of that interest are three invited presentations. Intel and AMD have invited Dr Scholz to present on the SaC compiler at Intel's European Research and Innovation Conference (ERIC 2010 in Braunschweig, Germany) and at AMD's Fusion Developer Summit 2011 (AFDS 2011 in Seattle, USA), respectively. Also Professor Jesshope has been invited to present on the Microgrid, also at Intel's European Research and Innovation Conference (ERIC 2011 in Leixlip, Ireland).

Embedded systems industry. Here we have been is discussion with a number of European companies as well a major European contractor. Typically these have involved visiting the company concerned, giving a seminar of the work followed by discussions on potential collaboration. The companies included ARM and NXP. NXP initially showed considerable interest following a seminar by Professor Jesshope and this was followed up by a HiPEAC Internship, where one of the Apple-CORE researchers spent 3 months on site at NXP. Unfortunately the recent reorganisation at NXP killed any further collaboration. The one success in this area has been with the European Space Agency. Professor Jesshope visited ESTEC at Noordwijk in the Netherlands and presented a seminar on the Apple-CORE results. There was considerable interest in the work, in particular in relation to the potential of the approach to make a considerable impact into real-time and reliability issues. This has resulted in a joint project being submitted to ESA's Innovation Triangle Initiative. This project *Multi-threaded processor for space applications* will directly exploit the results of the Apple-CORE project and investigate the feasibility of their use in space. The objective of the proposal is to investigate the benefit of multi-threaded processors in space applications. This includes generic applications as well as two specific areas which are critical for space missions: real-time tasks, where multi-threading can provide tighter bounds for WCET; and reliability, where multi-threading can provide more efficient support for redundant execution. A multi-threaded software SPARC emulator will be used to demonstrate these advantages.

Academic collaborations. Seminars have been given at a number of other universities and research institutions with the aim of establishing a wider community to adopt, evaluate and eventually exploit the APPLE-CORE results. This has included the Universities of Cambridge, York, St Andrews, Twente, Linköping and also INRIA, Institute of Computer Technology (ICT) Beijing and the Chinese Academy of Space Sciences (CAST) Beijing. These activities have resulted in a number of projects, project submissions and follow-up meetings where collaborative projects has been discussed. Tangible results from this dissemination activity include the ADVANCE project, now running, which includes the University of Twente and St. Andrews and which will adopt and adapt the SVP system-level model for resource management and will exploit the SaC compiler. Follow up meetings with ICT in Amsterdam, where the synergy with the work undertaken in the APPLE-CORE project has been discussed in detail. This has now been reflected in terms of a supported International Visiting Professorship at ICT for Professor Jesshope, to be taken up in 2012, during which a new project in Elastic Computer Architecture will be discussed and fleshed out. This will be based on the results of APPLE-CORE project and ongoing work at ICT. Another ongoing developments is a FET project submission, in preparation with the University of York and the University of Limerick, which will adopt both the Microthread model for programming and SVP system level model but apply these to stack-based architectures, which have been extensively researched at York. Finally collaborations with the University of Linköping on using SAC with the OpenModelica compiler. The use and promotion of Unibench with international compiler groups in industry and academia and collaboration within the HiPEAC project.

Section 3 - Simulated Architecture

The evaluation was performed from two perspectives:

- evaluation of the architecture design with regards to fixed software;
- evaluation of the software tool chain with regards to fixed architecture parameters.

To factor the effort required, the research was split in three phases:

1. *prior* to the benchmarking effort, a first architecture design analysis was carried out and produced a spectrum of "sane" parameters which reflect possible configurations of the architecture using current and future technology. As a result, two *architecture profiles* were designed which correspond to the ends of the identified spectrum. These were named the "128" and "256" architectures for labeling experiments; their parameters are described in Section 3.2 below.
2. most software benchmarks were evaluated against both profiles. These are reported in the rest of this report.
3. at the end of the evaluation period, further analysis of the architecture parameters were carried out based on the benchmark results. Although this is not part of the tasks defined by the DoW, these findings are described below in Section 3.3. As a result, a new "128" machine profile was designed with adjusted parameters. These are also described in Section 3.2. They are further used for the measurements in Section 4.

3.1 Understanding clock cycle reports

The emulation environment is based on an event-driven simulation of components using different clock frequencies. To organize the synchronization across frequency domains, the simulation uses an internal "master" clock which is an integer factor greater than every clock frequency in the system, ie the lowest common multiple of all clock frequencies.

In the benchmark results, the results are reported using cycle counts from this virtual master clock. This may create a difficulty when comparing results from different architecture parameters, because a small change in a clock frequency can cause large changes in the master clock frequency. For example when the core frequency increases from 1GHz to 1.2GHz, and the DDR3 frequency increases from 800MHz to 1.2GHz, the master frequency *decreases* from 4GHz to 1GHz. To ease conversion to normalized (comparable) results, the architecture descriptions below specify both the core frequency, the master frequency, and the factor to apply to the measured clock cycles to obtain a measurement in terms of core cycles.

3.2 Machine profiles

	"128" profile	"256" profile	new "128" profile
Number of cores	128	256	128
Number of FPUs	64	128	64
Core/FPU frequency	1GHz	1.2GHz	1GHz
L1 cache size (I/D)	1K/1K	1K/1K	2K/4K
Number of L2 caches	32	64	32
L2 cache size	32K	32K	128K
Number of DDR channels	2	4	4
DDR timings	11-11-11-28	9-11-9-27	11-11-11-28
DDR frequency	800MHz	1.2GHz	800MHz
Master frequency	**4GHz**	**1.2GHz**	**4GHz**
Master-to-core frequency ratio	**4**	**1**	**4**

The DDR parameters for both “128” profiles are suitable for simulation of realistic DDR3-1600 timings from the industry. The DDR parameters for the “256” profile are suitable for DDR3-2400 timings.

3.3 Evolution of the “128” machine profile

The initial design for the 128-core microgrid used previous research results [JLZ09] which suggested conservative settings for L1 D-cache size and flexible settings with the L2 cache size. The L2 cache size was initially scaled to 32K after observing low memory access rates and high cache reuse rates across a small selected range of compiler test cases.

During the project, it appeared that actual memory access rates were higher than initially expected. Cache misses were causing eviction traffic on the COMA rings, in turn reducing available bandwidth and preventing enough data to arrive to the cores to keep them busy with work. After further analysis, it was concluded that increasing the L2 cache size from 32K to 128K would be technically possible with current technology for a total of 8MB L2 cache on-chip (only with 128 cores). This was implemented in the new “128” profile in the last phase of the project.

Another significant change concerns the number of DDR channels. The external interface was initially configured to use only 2 DDR channels, considering the industry standards at the start of the project. Since then the Intel Core i5 and i7 designs have demonstrated the feasibility of 3 on-chip DDR controllers with enough unused pins on the package to support a 4th channel. It should be noted that the pinout on the package is the strongest constraint to increased channel counts, as each channel requires between 200 and 250 pins. To reflect this technology advance the new “128” profile is configured to use 4 DDR channels.

Finally, the L1 cache size was also adjusted. This stemmed from the observation that asynchrony of processing between memory accesses and other operations on the microthreaded core is only possible if a L1 cache line is reserved for pending memory accesses. When running concurrent heterogeneous workloads on a core (such as when running multiple application components on the same core simultaneously, or using functional concurrency), L1 cache reuse is diminished, asynchrony is limited and the number of pipeline stalls increase even though there are enough threads to potentially tolerate the latency. It was determined through experiments that a 2K I-cache was sufficient to sustain the heterogeneous instruction load from our benchmarks and a 4K D-cache was sufficient to tolerate the incurred memory activity. Further research on cache parameters is planned as future work.

Section 4 - Benchmarks

In the following subsections the benchmarks used within the Apple-Core project are split into groups and looked at one group at a time. In each case first the group is introduced with some explanation of the benchmarks and their relevance to the Apple-Core project. Then experimental results are provided with some analysis for one or more benchmarking experiments.

4.1 Livermore Loops

Livermore loops are benchmarks for parallel computers. They were created at Lawrence Livermore National Laboratory in 1986 [McM86].

These loops contain a mix of both local and global memory access patterns. They include both independent and dependent loops, which need to be recognised and mapped to the restrictions on the microgrid's shared register communication and synchronisation as well reductions which can be recognised and reordered. The more complex loops will also challenge the low-level compiler in optimising the use of register resources.

These sorts of simple loop are particularly good for the low level languages in the project, namely C and SL. For high-level languages like SaC optimisation techniques like With-loop fusion [GHS06] and With-loop scalarisation [GST04] help lower the total number of these loops and control the granularity of workloads. Consequently explicitly coding these in SaC can be articificial and attempts to do this often result in code with extra memory allocations and operations that don't pay off until the complexity of applications rise. Despite this however it is still the case that the Sac2c compiler is expected to parallelise some of these patterns on and off of the microgrid. In these cases the scaling of the performance results on increasing cores is analysed.

As a set, they represent a range of kernels that will challenge any parallelising compiler. These loops contain a mix of both local and global memory access patterns. They include both independent and dependent loops, which need to be recognised and mapped to the restrictions on the microgrids shared register communication and synchronisation as well reductions which can be recognised and reordered. The more complex loops will also challenge the low-level compiler in optimising the use of register resources.

The latest graphs and results for experiments in this section can be found in Unibench at https://unibench.apple-core.info/?page=benchmark_suites&filterId=106

4.1.1 Livermore Loop: 1

Kernel 1 – hydro fragment

```
      DO 1 L = 1,Loop
      DO 1 k = 1,n
1        X(k)= Q + Y(k)*(R*ZX(k+10) + T*ZX(k+11))
```

Figure 1 shows the SL results. They represent the ideal results since they are hand-coded examples written in the language that the SaC and SL toolchains compile towards. The SL results scale to the full 64 cores. Note that as explained in Section 3.2 the clock cycles shown here are ‘master clock cycles.’ The SL hand-coded examples scale from approximately 65 thousand core clock cycles to 12 thousand cycles.

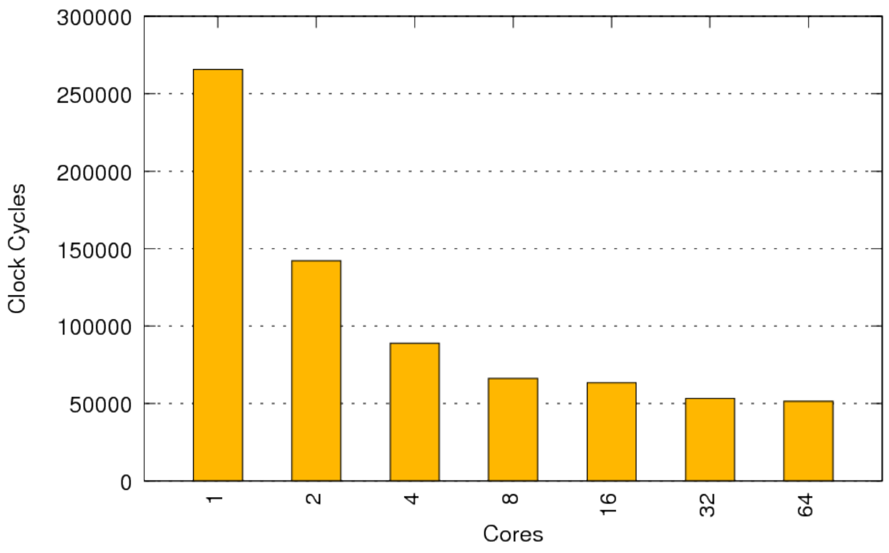


Figure 1: Simulated results for the SL implementation of the Livermore Loop 1 benchmark on the 128 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=698)

Figure 2 shows the SaC results and Figure 3 shows the C results. All three toolchains show a speedup onto all 64 cores like the hand-coded SL results.

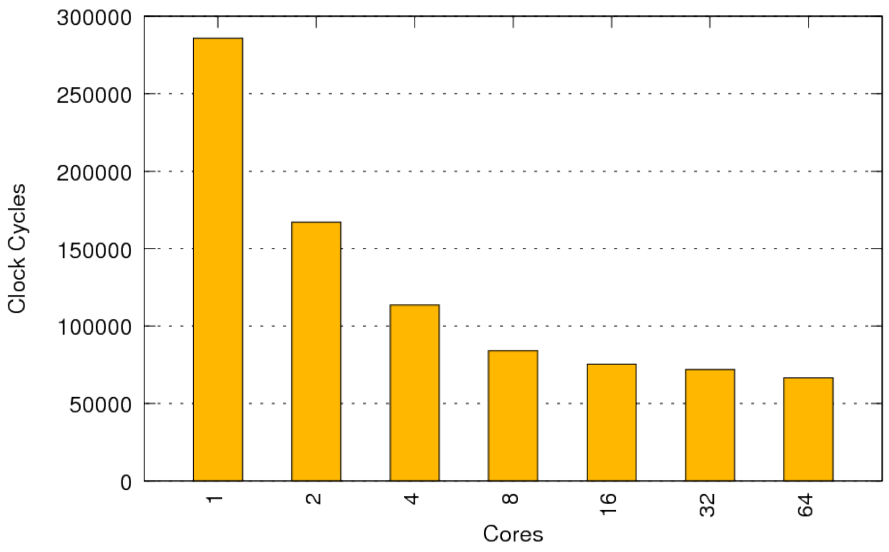


Figure 2: Simulated results for the SaC implementation of the Livermore Loop 1 benchmark on the 128 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=702)

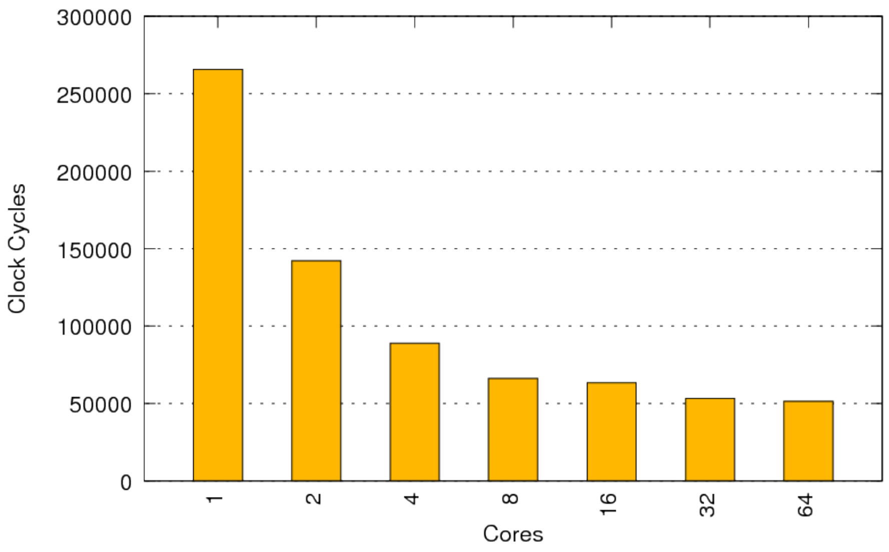


Figure 3: Simulated results for the C implementation of the Livermore Loop 1 benchmark on the 128 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=703)

The SaC results are consistently 3K master cycles slower than the ideal SL results. We investigated to find out why. We discovered that this difference stems from two sources.Firstly, it is inherent to the high-level approach of SaC that the generated code contains memory management overheads such as result allocations and dynamic re-use checks whereas the hand-coded SL examples could be specifically written to measure just the loop itself without any of that administration overhead. Another source for the runtime difference stems from the way SaC presents complex expressions to the SL compiler, namely, by means of sequences of assignments to intermediate variables. While the SL compiler should be able to handle these equally well, it turns out that it typically generates slightly less compact code. However, considering real hardware and more realistic problem sizes (which we could not run here due to simulation time restrictions) a static difference of 3000 cycles seem insignificant.

The small measured overheads for SaC are typical and will commonly be exposed in tiny applications on small data sttructures like small Livermore Loop examples optimised for hardware simulation for example. For the remaning loops these details will not be mentioned.

For this loop good scaling and comparable performance was achieved for all three toolchains.

4.1.2 Livermore Loop: 3

Livermore Loop 3 is defined as:

$$\sum_i x_i y_i$$

```
     Kernel 3 -- inner product

     DO 3 L= 1,Loop
          Q= 0.0
     DO 3 k= 1,n
   3      Q= Q + Z(k)*X(k)
```

In Figure 4 this kernel implements what functional programmers call a fold: A binary operation is performed on all adjacent elements of a vector to produce a single scalar result. For example a fold of the plus operation on a list of integers is the sum operation that adds up all the numbers in the vector. Our naive parallel implementation splits the vector evenly into segments and ‘folds’ each to a scalar in linear time before finally folding the results from each segment.

With this kernel in SaC we looked at the potential benefits of implementing a logarithmic based fold to improve upon our naive linear algorith.

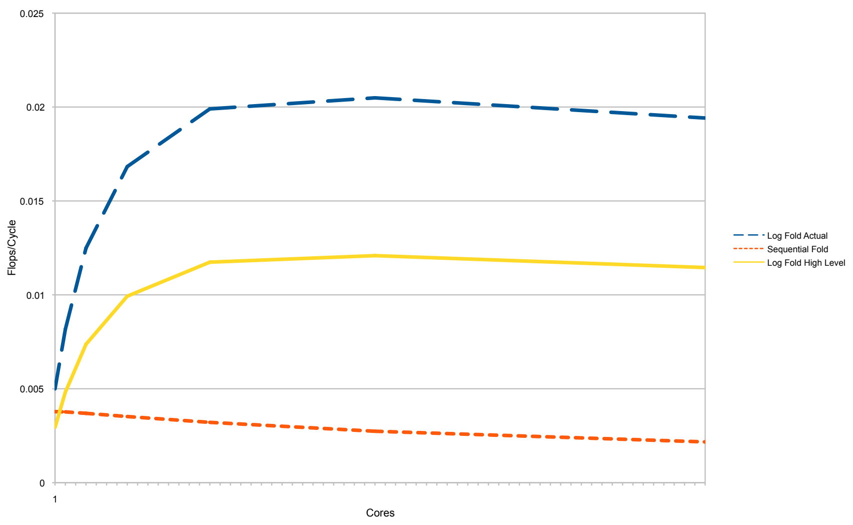


Figure 4: The simulated Livermore Loop 3 kernel implemented in SaC.

In Figure 4 the floating point operations per cycle are compared between our sequential and parallel versions. Our parallel algorithm increases the number of floating point operations and so for fair comparisson we show both actual number of calculations (‘Log Fold Actual’) and the same calculation using the ideal number of floating point calculation which is the number used for the sequenial version (‘log fold high level’). As can be seen in Figure 4 the logarithmic algorithm increases the number of floating point operations that are needed to compute the reduction. However even though there are more floating point operations because of the greater scalability of this method we can see that the addition of floating point operations pays off though faster throughput.

Figure 5 shows how this looks in terms of runtime as the number of cores is increased for handcoded SL code.

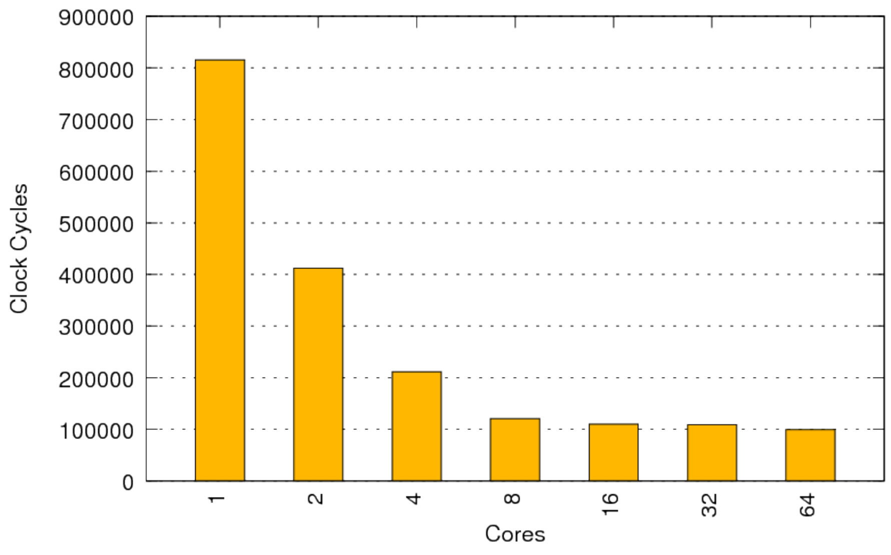


Figure 5: Simulated results for the SL implementation of the Livermore Loop 3 benchmark on the 128 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=729)

Figure 6 is an artificial example written in SaC to show how scaling would improve once a logarithmic parallel fold is implemented.

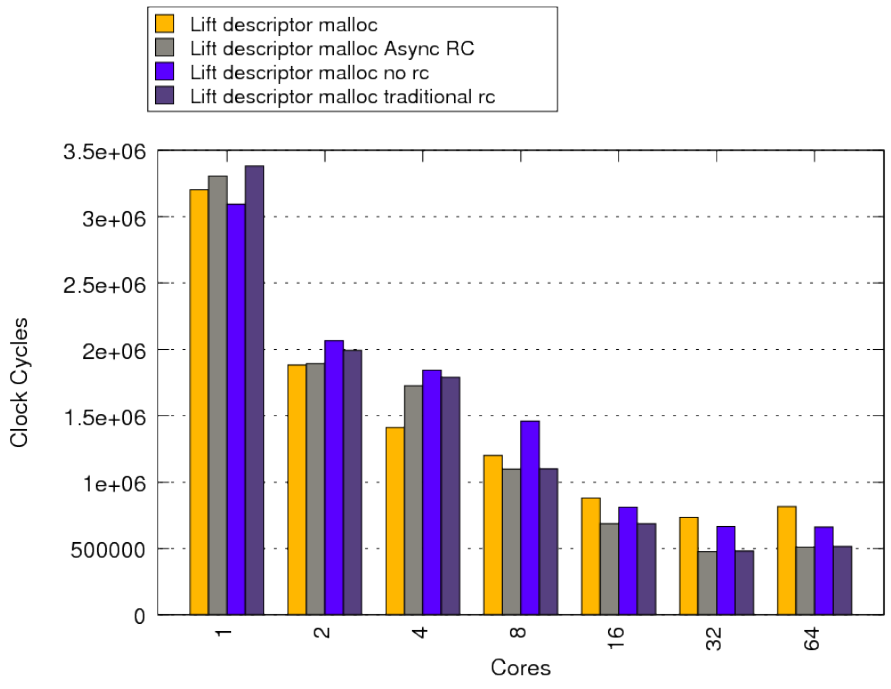


Figure 6: Simulated results for the SaC implementation of the Livermore Loop 3 benchmark on the 256 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=726)

4.1.3 Livermore Loop: 4

Kernel 4 – banded linear equations

```
                 m= (1001−7)/2
      DO 444     L= 1,Loop
      DO 444     k= 7,1001,m
                lw= k−6
              temp= X(k−1)
CDIR$ IVDEP
      DO   4   j= 5,n,5
          temp   = temp    − XZ(lw)*Y(j)
  4             lw= lw+1
          X(k−1)= Y(5)*temp
444 CONTINUE
```

Figure 7 shows results of hand-coded SL code for linear equations on banded matricies. These results depend on an efficient fold algorithm and the SaC Livermore Loop 3 results suggest that once this is implemented that similar speedup could easily be demonstrated from application code written in SaC.

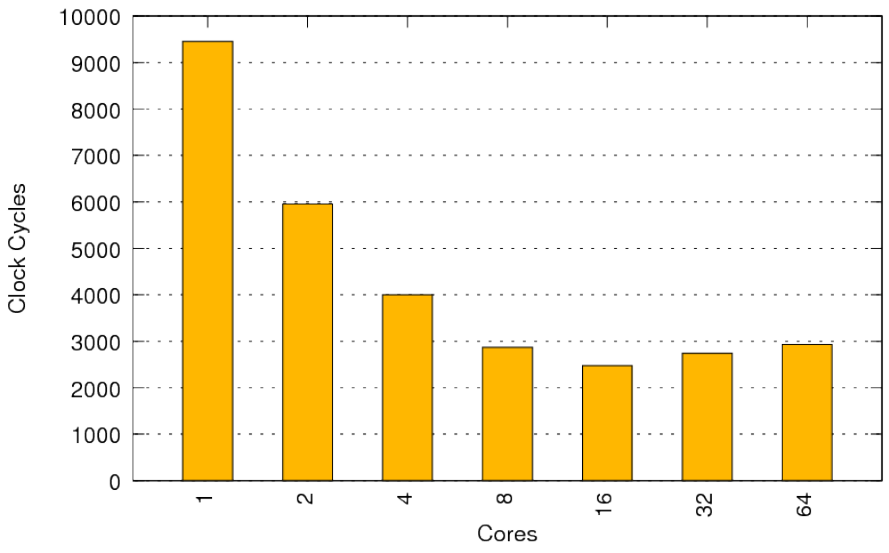


Figure 7: Simulated results for the SL implementation of the Livermore Loop 4 benchmark on the 256 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=653)

4.1.4 Livermore Loop: 7

Kernel 7 – equation of state fragment

```
      DO 7  L=  1,Loop
      DO 7  k=  1,n
        X(k)=     U(k  ) + R*( Z(k  ) + R*Y(k  )) +
     .         T*( U(k+3) + R*( U(k+2) + R*U(k+1)) +
     .         T*( U(k+6) + R*( U(k+5) + R*U(k+4))))
    7 CONTINUE
```

Figure 8 shows the results of the hand-coded SL implementation. Good scaling is achieved over all cores.

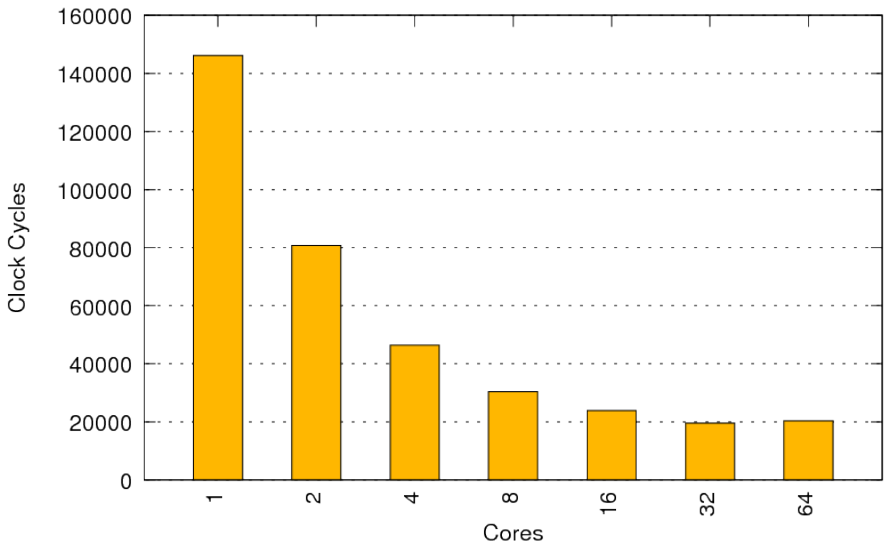


Figure 8: Simulated results for the SL implementation of the Livermore Loop 7 benchmark on the 128 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=655)

Figure 9 shows the results for the C implementation. The results almost match the SL implementation except that on high core counts the runtimes are actually slightly faster than even the hand-coded examples which is a great and unexpected achievement on small, independent loops.

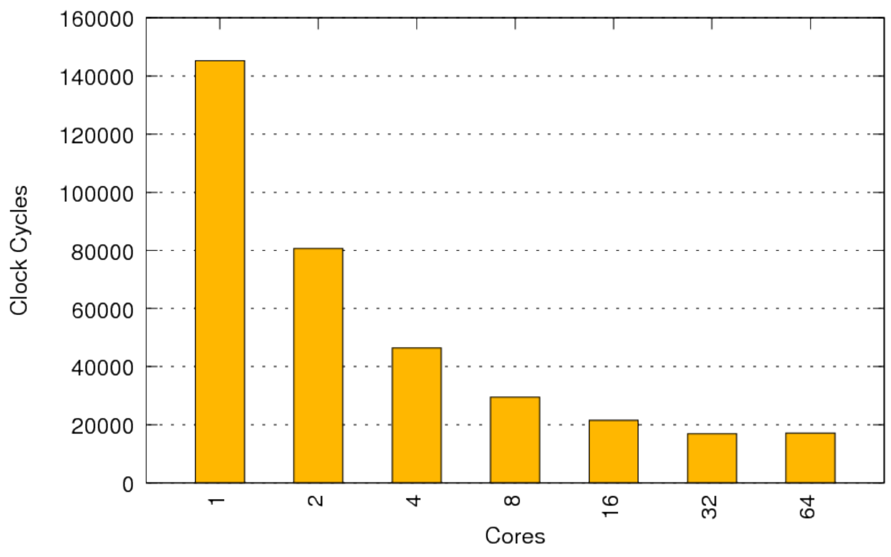


Figure 9: Simulated results for the C implementation of the Livermore Loop 7 benchmark on the 128 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=687)

Figure 10 shows the SaC results which still show a very similar speedup when initial overhead for measuring is taken into account.

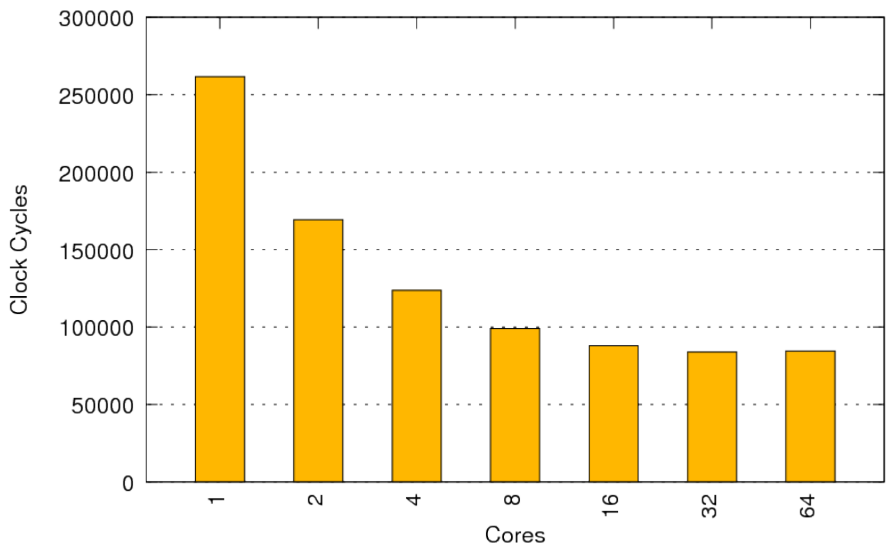


Figure 10: Simulated results for the SaC implementation of the Livermore Loop 7 benchmark on the 128 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=657)

4.1.5 Livermore Loop: 8

Kernel 8 – ADI integration

```
      DO   8      L = 1,Loop
                nl1 = 1
                nl2 = 2
      DO   8     kx = 2,3
CDIR$ IVDEP
      DO   8     ky = 2,n
              DU1(ky)=U1(kx,ky+1,nl1)  -  U1(kx,ky-1,nl1)
              DU2(ky)=U2(kx,ky+1,nl1)  -  U2(kx,ky-1,nl1)
              DU3(ky)=U3(kx,ky+1,nl1)  -  U3(kx,ky-1,nl1)
       U1(kx,ky,nl2)=U1(kx,ky,nl1) +A11*DU1(ky) +A12*DU2(ky) +A13*DU3(ky)
     .       + SIG*(U1(kx+1,ky,nl1) -2.*U1(kx,ky,nl1) +U1(kx-1,ky,nl1))
       U2(kx,ky,nl2)=U2(kx,ky,nl1) +A21*DU1(ky) +A22*DU2(ky) +A23*DU3(ky)
     .       + SIG*(U2(kx+1,ky,nl1) -2.*U2(kx,ky,nl1) +U2(kx-1,ky,nl1))
       U3(kx,ky,nl2)=U3(kx,ky,nl1) +A31*DU1(ky) +A32*DU2(ky) +A33*DU3(ky)
     .       + SIG*(U3(kx+1,ky,nl1) -2.*U3(kx,ky,nl1) +U3(kx-1,ky,nl1))
    8 CONTINUE
```

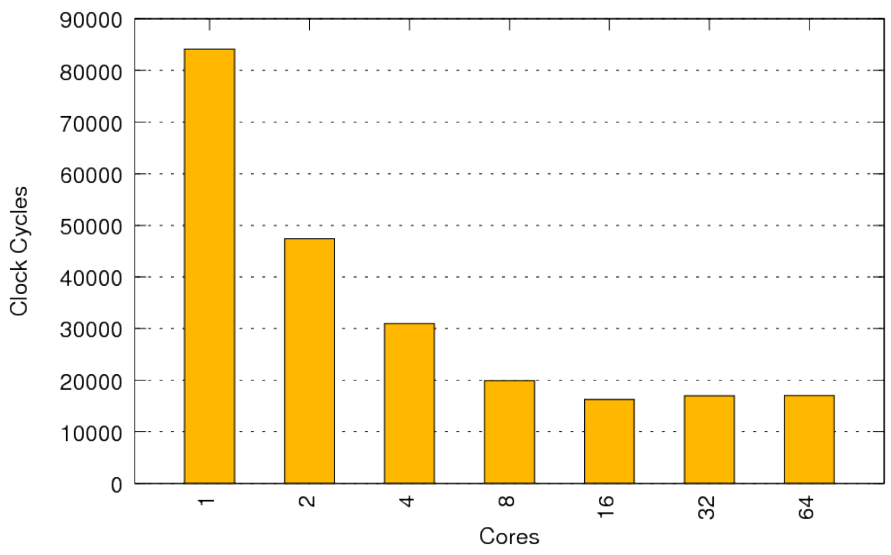


Figure 11: Simulated results for the SL implementation of the Livermore Loop 8 benchmark on the 128 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=664)

Figure 11 shows the hand-coded SL results for Livermore Loop 8. Initally near-linear scaling is exhibited and scaling continues until 16 cores.

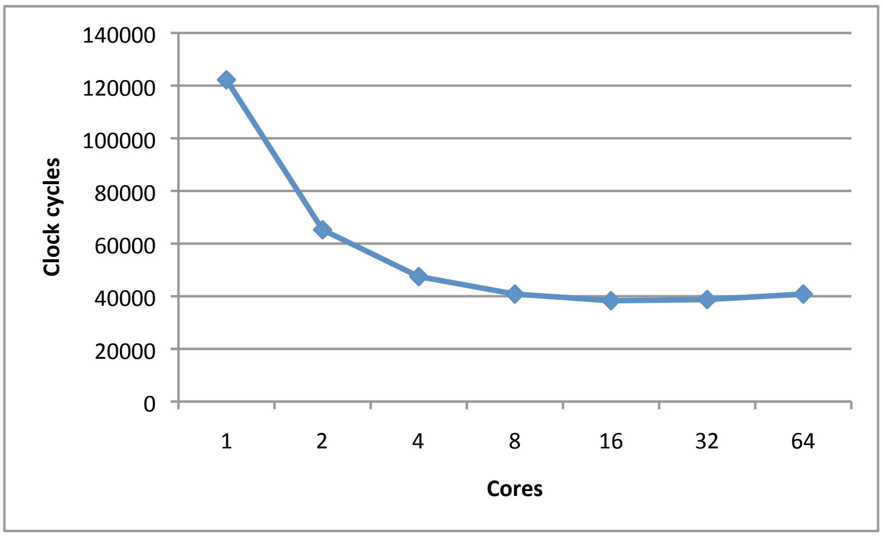


Figure 12: Simulated results for the SAC implementation of the Livermore Loop 8 benchmark on the 128 architecture.

Figure 12 shows the SAC results. SAC2Cis able to fuse together the loops for all six assignments for an optimal memory access pattern and achieves the same scaling as the hand-coded SL version with just a small overhead which we put down to measuring accuracy.

4.1.6 Livermore Loop: 9

Kernel 9 – integrate predictors

```
   DO 9   L  =  1,Loop
   DO 9    i  =  1,n
   PX( 1,i)= DM28*PX(13,i) + DM27*PX(12,i) + DM26*PX(11,i) +
  .           DM25*PX(10,i) + DM24*PX( 9,i) + DM23*PX( 8,i) +
  .           DM22*PX( 7,i) +   C0*(PX( 5,i) +       PX( 6,i))+ PX( 3,i)
9 CONTINUE
```

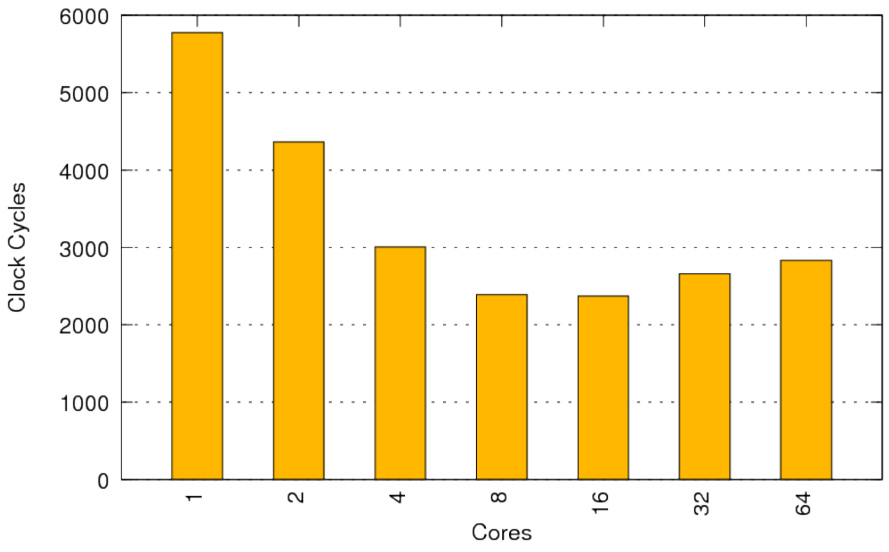


Figure 13: Simulated results for the SL implementation of the Livermore Loop 9 benchmark on the 256 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=688)

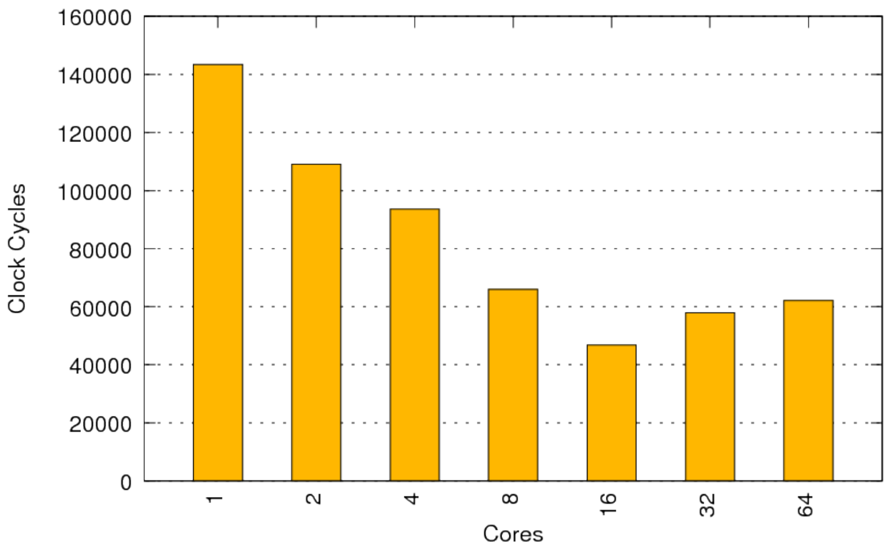


Figure 14: Simulated results for the SAC implementation of the Livermore Loop 9 benchmark on the 256 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=686)

4.1.7 Livermore Loop: 10

Kernel 10 – Difference predictors

```
/*
 *******************************************************************
 *    Kernel 10 -- difference predictors
 *******************************************************************
 *     DO 10   L=  1,Loop
 *     DO 10   i=  1,n
 *     AR        =       CX(5,i)
 *     BR        = AR - PX(5,i)
 *     PX(5,i)  = AR
 *     CR        = BR - PX(6,i)
 *     PX(6,i)  = BR
 *     AR        = CR - PX(7,i)
 *     PX(7,i)  = CR
 *     BR        = AR - PX(8,i)
 *     PX(8,i)  = AR
 *     CR        = BR - PX(9,i)
 *     PX(9,i)  = BR
 *     AR        = CR - PX(10,i)
 *     PX(10,i)= CR
 *     BR        = AR - PX(11,i)
 *     PX(11,i)= AR
 *     CR        = BR - PX(12,i)
 *     PX(12,i)= BR
 *     PX(14,i)= CR - PX(13,i)
 *     PX(13,i)= CR
 *  10 CONTINUE
 */
```

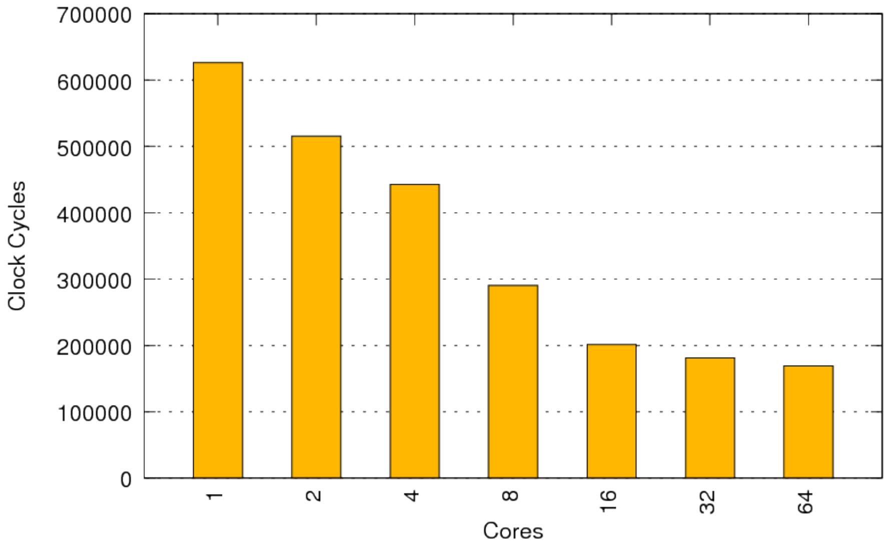


Figure 15: Simulated results for the SAC implementation of the Livermore Loop 10 benchmark on the 256 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=716)

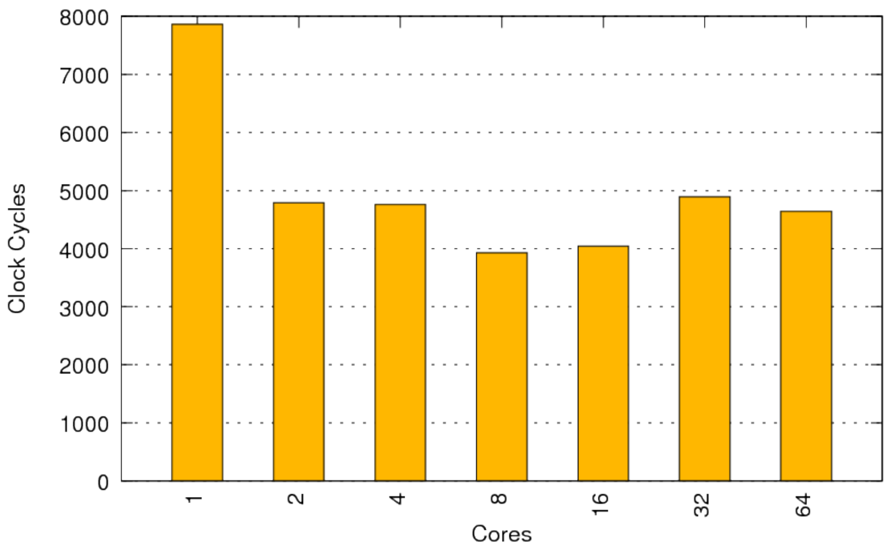


Figure 16: Simulated results for the SL implementation of the Livermore Loop 10 benchmark on the 256 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=717)

4.1.8 Livermore Loop: 12

Kernel 12 – first difference

```
      DO 12 L = 1,Loop
      DO 12 k = 1,n
12       X(k)= Y(k+1) - Y(k)
```

Figure 17 shows the hand-coded SL implementation.

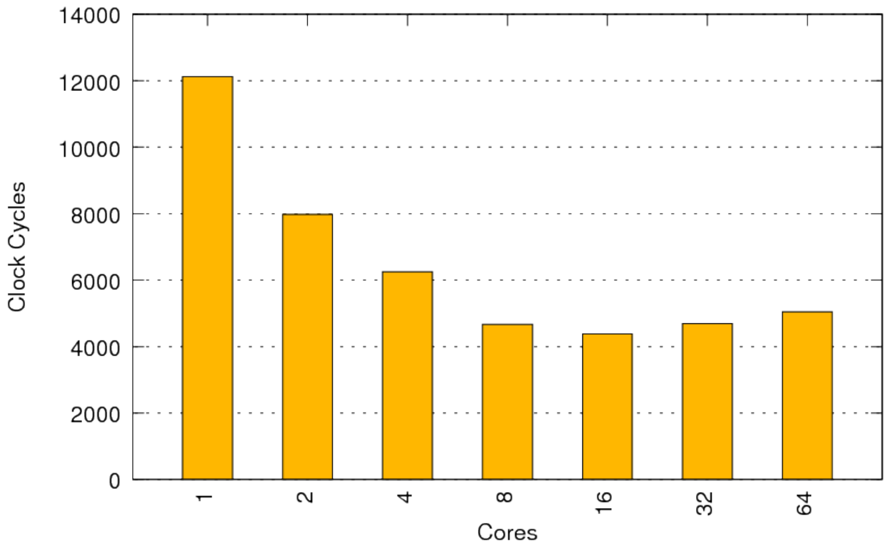


Figure 17: Simulated results for the SL implementation of the Livermore Loop 12 benchmark on the 256 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=692)

Figure 18 shows the C implementation which has very similar scaling to the hand-coded SL results which marginally bettre better performance.

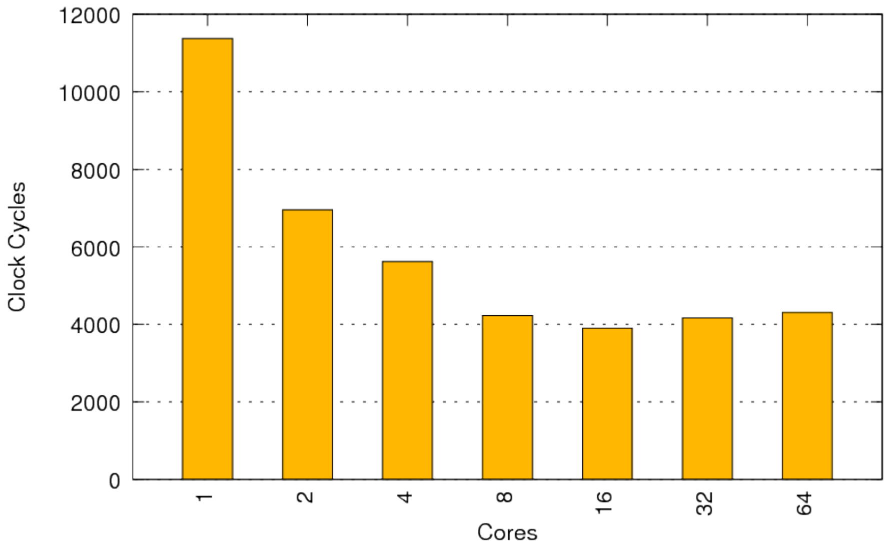


Figure 18: Simulated results for the C implementation of the Livermore Loop 12 benchmark on the 256 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=643)

Figure 19 shows the SaC implementation which again show very similar scaling but show slightly higher runtimes which is expected since runtimes for SaC are commonly shown as slighly higher than the actual runtimes as was discovered in Livermore Loop 1.

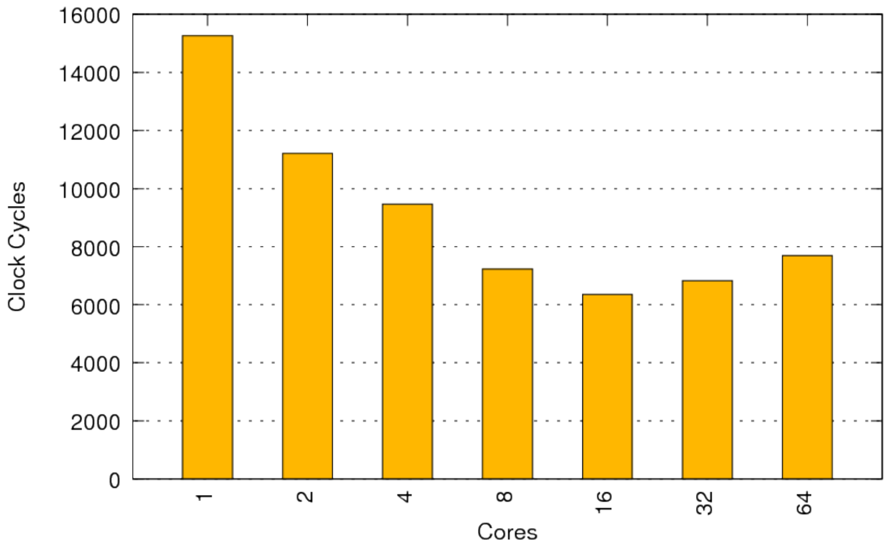


Figure 19: Simulated results for the SaC implementation of the Livermore Loop 12 benchmark on the 256 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=691)

4.1.9 Livermore Loop: 15

Kernel 15 – Casual Fortran. Development version

```
       DO 45   L  =  1,Loop
               NG=  7
               NZ=  n
               AR=  0.053
               BR=  0.073
 15    DO 45   j  =  2,NG
       DO 45   k  =  2,NZ
               IF(  j-NG)  31,30,30
 30      VY(k,j)=  0.0
                    GO TO  45
 31            IF(  VH(k,j+1)  -VH(k,j))  33,33,32
 32              T=  AR
                    GO TO  34
 33              T=  BR
 34            IF(  VF(k,j)  -VF(k-1,j))  35,36,36
 35              R=  MAX(  VH(k-1,j),  VH(k-1,j+1))
                 S=  VF(k-1,j)
                    GO TO  37
 36              R=  MAX(  VH(k,j),      VH(k,j+1))
                 S=  VF(k,j)
 37      VY(k,j)=  SQRT(  VG(k,j)**2  +R*R)*T/S
 38            IF(  k-NZ)  40,39,39
 39      VS(k,j)=  0.
                    GO TO  45
 40            IF(  VF(k,j)  -VF(k,j-1))  41,42,42
 41              R=  MAX(  VG(k,j-1),  VG(k+1,j-1))
                 S=  VF(k,j-1)
                 T=  BR
                    GO TO  43
 42              R=  MAX(  VG(k,j),      VG(k+1,j))
                 S=  VF(k,j)
                 T=  AR
 43      VS(k,j)=  SQRT(  VH(k,j)**2  +R*R)*T/S
 45     CONTINUE
```

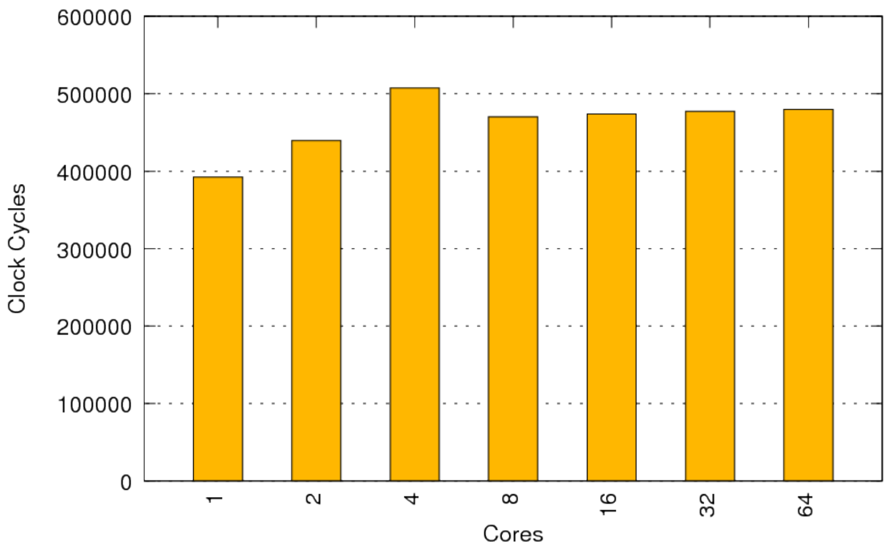


Figure 20: Simulated results for the SaC implementation of the Livermore Loop 15 benchmark on the 256 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=588)

4.1.10 Livermore Loop: 19

Kernel 19 – General linear recurrence equations

```
/*
 ******************************************************************
 *    Kernel 19 —— general linear recurrence equations
 ******************************************************************
 *                KB5I= 0
 *             DO 194 L= 1,Loop
 *             DO 191 k= 1,n
 *         B5(k+KB5I)= SA(k) +STB5*SB(k)
 *               STB5= B5(k+KB5I) -STB5
 *191           CONTINUE
 *192           DO 193 i= 1,n
 *                  k= n-i+1
 *         B5(k+KB5I)= SA(k) +STB5*SB(k)
 *               STB5= B5(k+KB5I) -STB5
 *193           CONTINUE
 *194 CONTINUE
 */
```

Figure 21 shows the results of the hand-coded SL implementation. This graph shows the initial run (cold cache) and then a second run (warm cache). Note that where not otherwise mentioned all measurements in this report are cold cache where the architecture has no means of taking advantage of data already in the cache. As with traditional architectures the Microgrid still takes advantage of caching when runnig code patterns in loops.

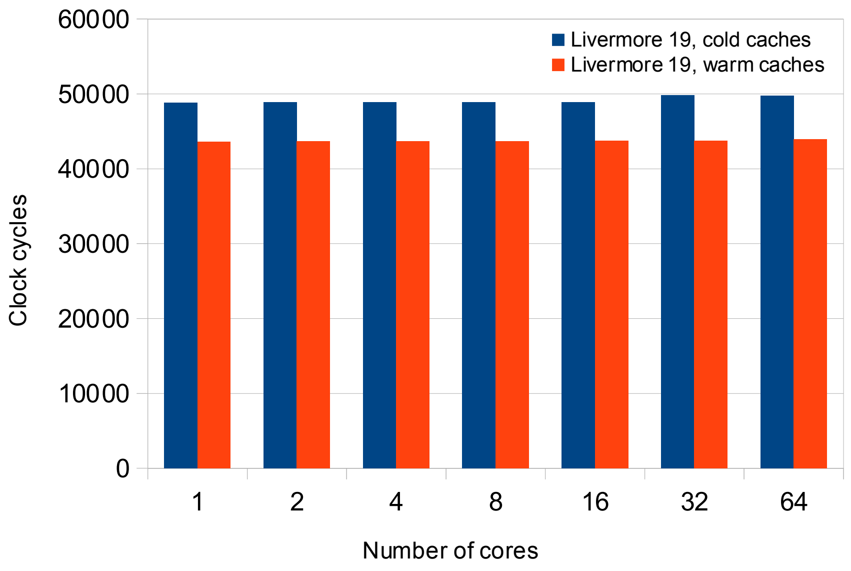


Figure 21: Simulated results for the SL implementation of the Livermore Loop 19 benchmark on the 256 architecture.

Figure 22 shows the same experiment run with the SAC toolchain. Initially the scaling is bad but on large numbers of cores the runtimes become noticably smaller than the runtimes in the hand-coded SL version. The dependencies in this loop severely limit parallelisation potential but small speedups are still achieved in the SAC example.

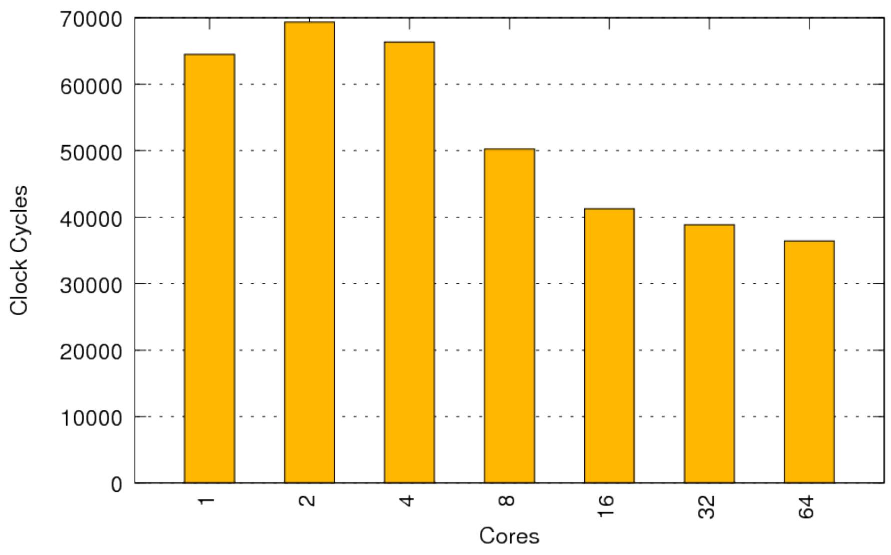


Figure 22: Simulated results for the SaC implementation of the Livermore Loop 19 benchmark on the 256 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=708)

4.1.11 Livermore Loop: 21

Kernel 21 – Matrix matrix product

```
/*
      *******************************************************************
      *    Kernel 21 -- matrix*matrix product
      *******************************************************************
      *     DO 21 L= 1,Loop
      *     DO 21 k= 1,25
      *     DO 21 i= 1,25
      *     DO 21 j= 1,n
      *     PX(i,j)= PX(i,j) +VY(i,k) * CX(k,j)
      * 21 CONTINUE
      */
```

This Livermore Loop is important since matrix multiplication is common in numerical codes. It is another variation of what was measured in Section 4.8. Figure 23 shows the hand-coded SL measurements. As with our other matrix multiplication experiments the scaling is very good.

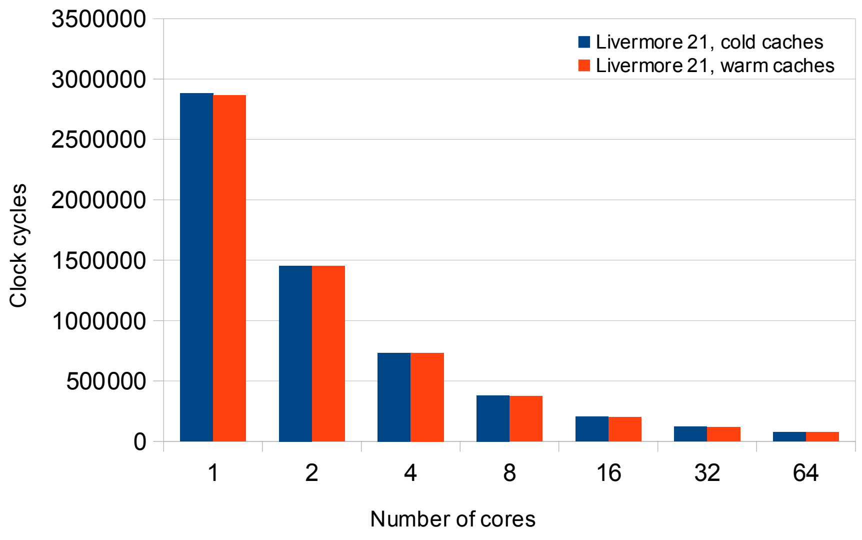


Figure 23: Simulated results for the SL implementation of the Livermore Loop 21 benchmark on the 256 architecture.

Figure 23 shows the same experiment in SaC the scaling is still good, although not as impressive as the hand-coded SL results but the overall runtime is considerably better.

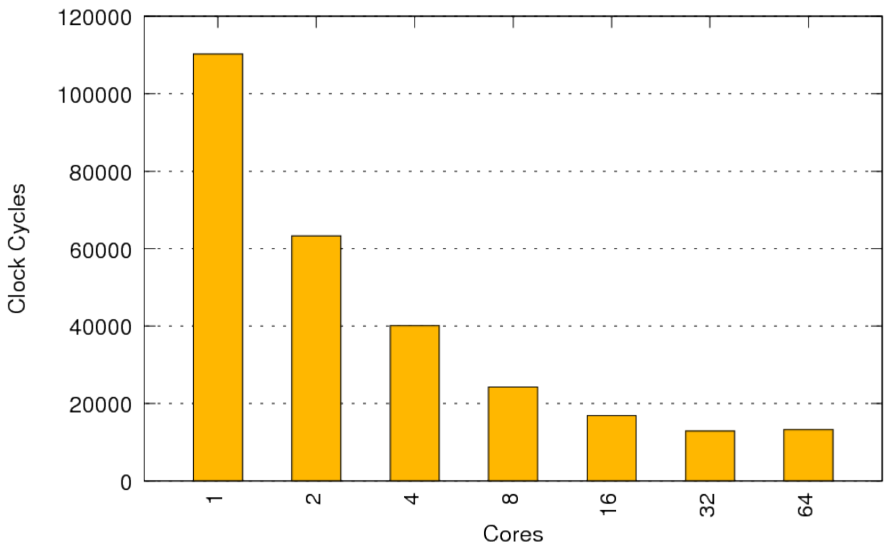


Figure 24: Simulated results for the SaC implementation of the Livermore Loop 21 benchmark on the 256 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=707)

A more detailed matrix multiplication case study can be found in Section 4.8.

4.1.12 Livermore Loop: 22

Kernel 22 – Planckian distribution

```
*******************************************************************
*    Kernel 22 —— Planckian distribution
*******************************************************************
*        EXPMAX= 20.0
*           U(n)= 0.99*EXPMAX*V(n)
*       DO 22 L= 1,Loop
*       DO 22 k= 1,n
*                                                   Y(k)= U(k)/V(k)
*           W(k)= X(k)/( EXP( Y(k)) -1.0)
*  22 CONTINUE
*/
```

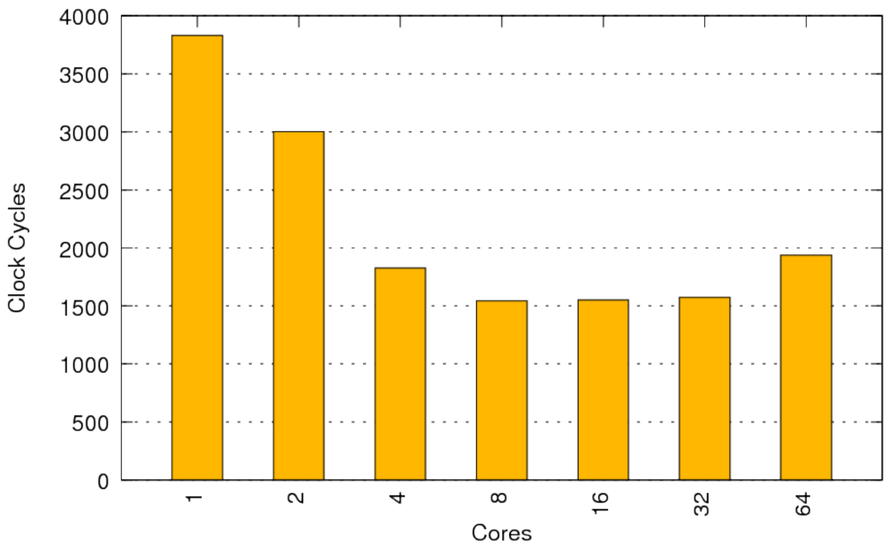


Figure 25: Simulated results for the C implementation of the Livermore Loop 22 benchmark on the 256 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=649)

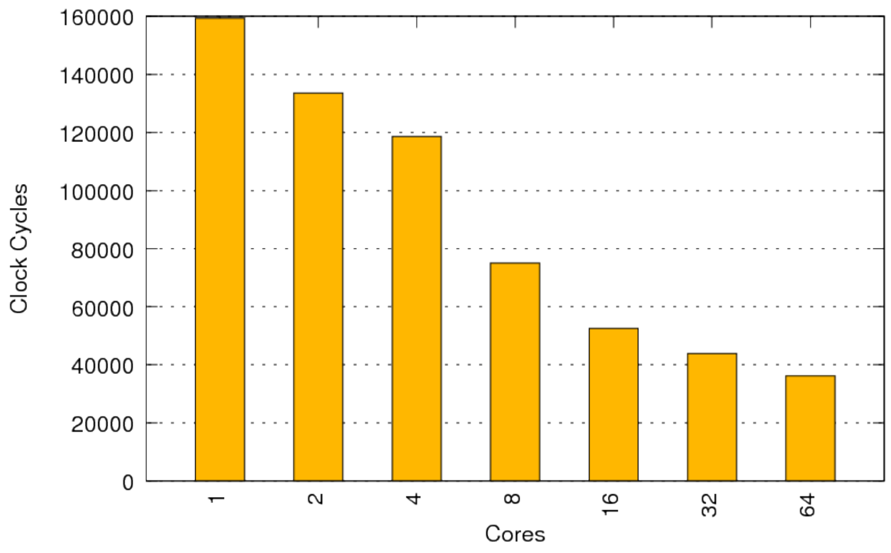


Figure 26: Simulated results for the SAC implementation of the Livermore Loop 22 benchmark on the 256 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=650)

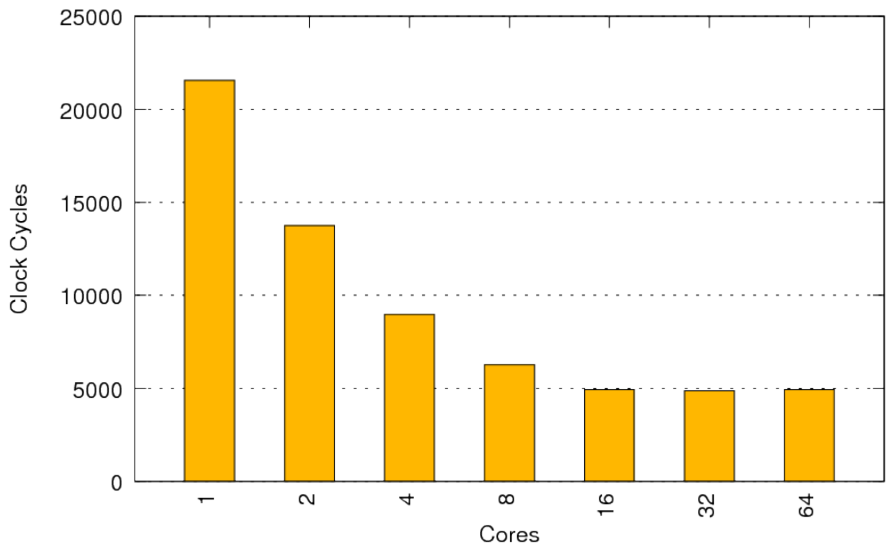


Figure 27: Simulated results for the SL implementation of the Livermore Loop 22 benchmark on the 256 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=689)

4.1.13 Conclusions

Clearly for small workloads of less than 1000 elements and the common scenarios in numerical code auto parallelisation can work well up to 32 cores with noticable gains up to 16 cores. As the problem size grows more cores can be efficiently utilised. These problems are known to be hard to parallelise but despite that autoparallelised SAC code often results in graphs looking incredibly similar to the hand-coded examples. The overhead added by SAC2C is often tiny andSAC results on legacy architectures have shown that these overheads go away in larger more complex, real-world applications.

l

4.2 N-Body Simulations

The N-body problem describes the evolution of a system of N bodies which comply to classical laws of mechanics. Each body interacts with all the other bodies. N-body algorithms have numerous applications in areas such as astrophysics, molecular dynamics and plasma physics [WS93]. Also, N-body is a part of the SPEC [nDYRHK06] benchmark suite and a Dwarf from Berkley's review paper [ABC$^+$06].

The classic implementation of an N-body simulation is of complexity $O(N^2)$ and is not suitable for large scale simulations. It is, however, widely used in benchmark suites for its simplicity. The existence of an N-body implementation on the Microgrid architecture is therefore imperative for direct comparison to classical architectures.

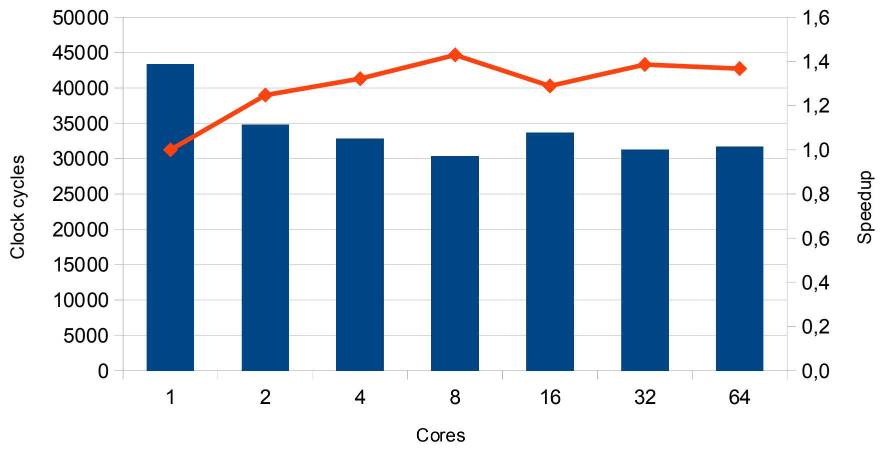


Figure 28: Simulated results for the SL implementation of the N-Body benchmark on the 256 architecture.

Figure 28 shows the results for the SL implementation of the algorithm. The simulation simulates 25 bodies in a 3D space advanced for just one timestep. The positions and accelerations of all 25 bodies are updated with respect to their positions and masses. The algorithm simply interacts on N by N by 2 arrays of double precision floating points representing position and acceleration vectors.It runs with a reasonable speed on one core completing in 43,000 cycles. For this very small problem size of just 25 bodies the scaling is poor.

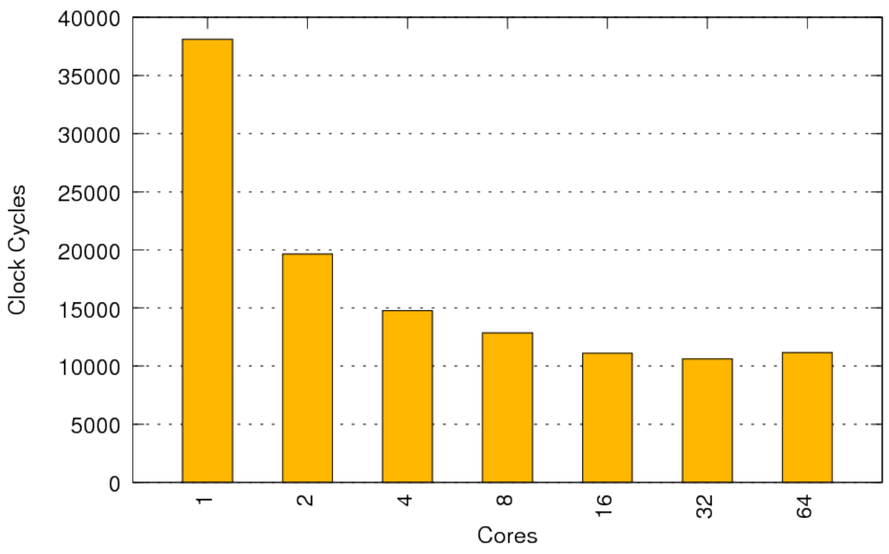


Figure 29: Simulated results for the SAC implementation of the N-Body benchmark on the 256 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=640)

The results of experiments on a SaC implementation of the classic algorithm are shown in Figure 29. This simple, highly understandable representation heeds linear speedup from 1 to 2 cores and continues with a much smaller speedup onto 13 cores.

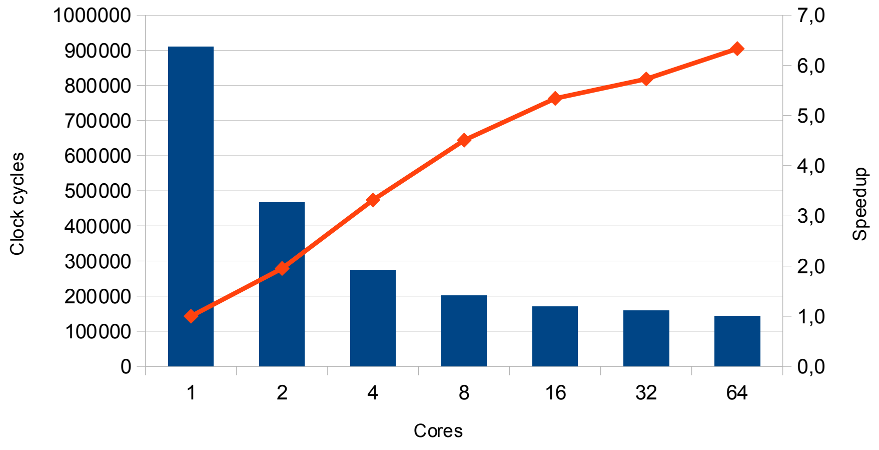


Figure 30: Simulated results for the SL implementation of the N-Body benchmark on the 256 architecture.

Figure 30 shows the SL results for an increased problem size of 100 bodies. This is enough to improve the speedup so that it continues up to the full 64 cores and speeds up to over 6 times.

The latest graphs and results for experiments in this section can be found in Unibench at
https://unibench.apple-core.info/?page=benchmark_suites&filterId=43

4.3 FFT

FFT algorithms compute Discrete Fourier Transforms and have applications in a whole host of other fields from signal processing to data compression. In particular they are important in many computational fluid dynamics applications. This benchmark gives us the opportunity to test implementations with known global memory access patterns.

Figure 31 shows a SaC simulation of a one dimensional FFT. Speedup as the number of cores is increased is significant which is pleasing since FFTs are incredibly well known and important in computational science.

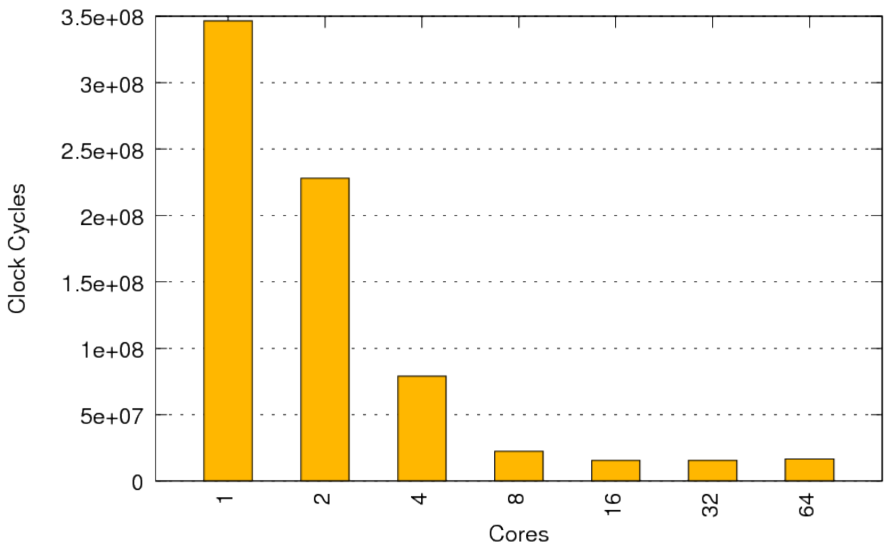


Figure 31: Simulated results for the SaC implementation of the One-dimensional FFT benchmark on the 256 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=638)

Figure 32 shows that the speedup peaks in excess of 22 times the runtime on a single core.

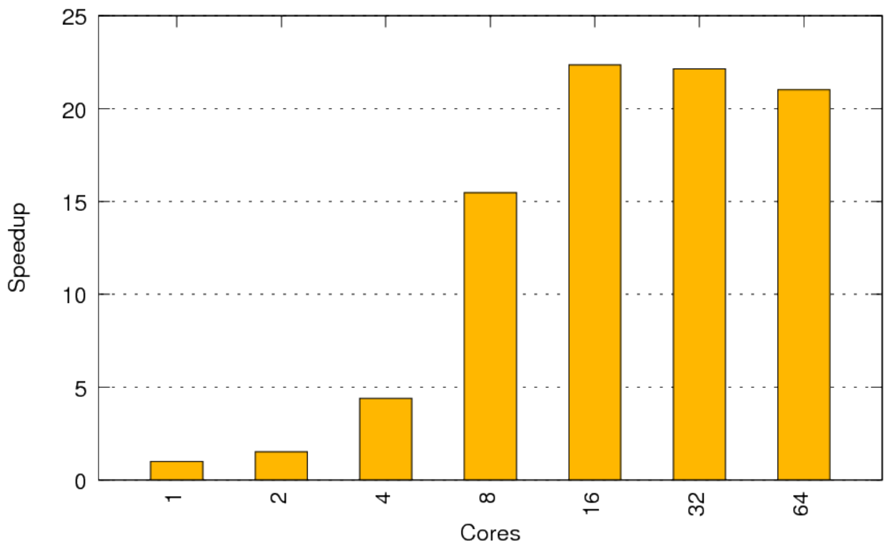


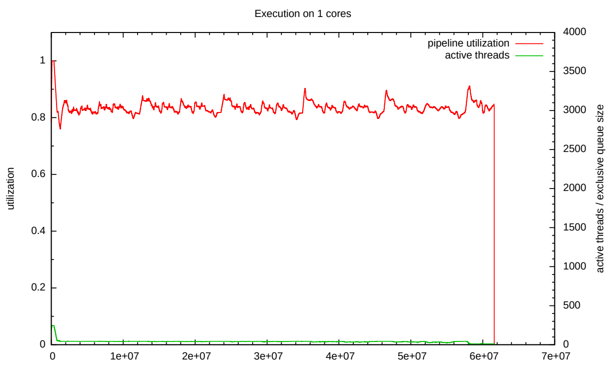
Figure 32: Simulated results for the SaC implementation of the One-dimensional FFT benchmark on the 256 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=696)

The latest graphs and results for experiments in this section can be found in Unibench at https://unibench.apple-core.info/?page=benchmark_suites&filterId=55

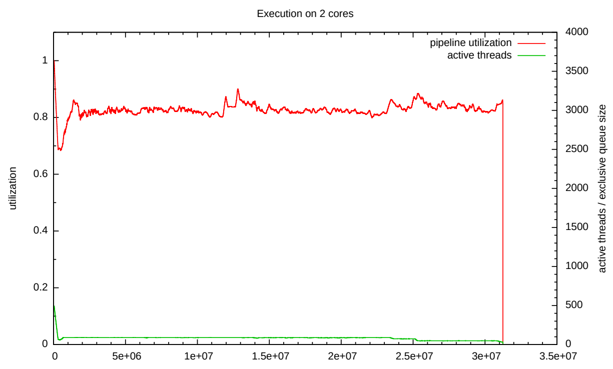
4.3.1 2D FFT

This Section briefly shows the 2-dimensional FFT results from Apple-Core Deliverable 4.4.

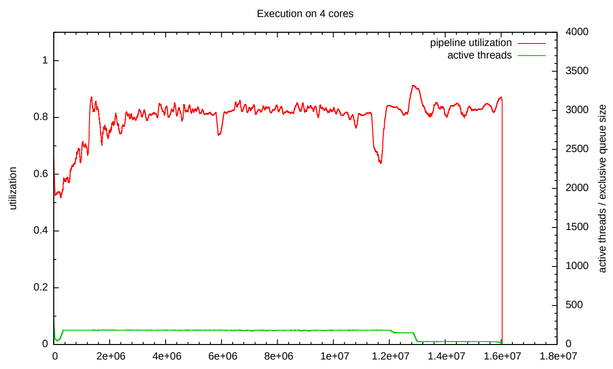
Figure 33 demonstrates the runtime behaviour of our approach for a two dimensional FFT on 256 element vectors. Linear scaling can be seen up to graph (c). This is analysed in more detail in Deliverable 4.4.



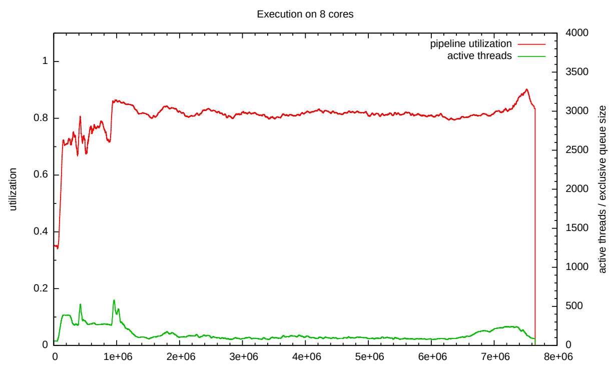
(a) one core



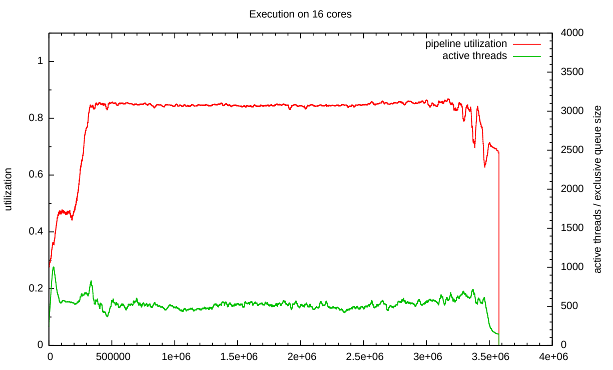
(b) two cores



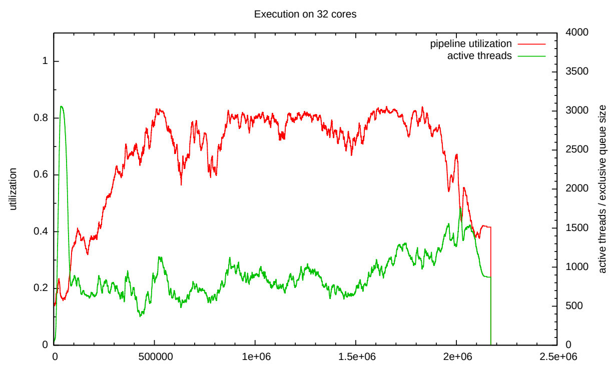
(c) four cores



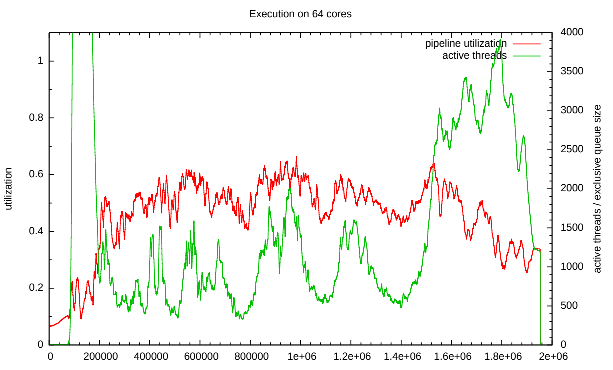
(d) 8 cores



(e) 16 cores



(f) 32 cores



(g) 64 cores

Figure 33: Runtime behaviour of a 2-dimensional 256 element FFT running on varying numbers of cores using asynchronous deferred reference counting.

4.4 Image Smoothing

The Image Smoothing benchmark is defined as:

$$I_{[i,j]} = \frac{\left(\sum_{a=i-1}^{i+1} \sum_{b=j-1}^{j+1} I_{[a,b]}\right) - I_{[i,j]}}{8}$$

Figure 34 shows floating point operations per cycle for core counts ranging from 1 to 64. The graph shows both simulated results on the 256 architecture and non-simulated results on a 64-core Opteron machine side by side. The non-simulated results are shown as 'Sac2mt' where mt stands for multi-threading and refers to code generated with SAC2C's PThread backend. In this graph it can be seen that the super scalar cores of an AMD Opteron are capable of a higher throughput than the cores of the Microgrid. However as the number of cores increase the ability of the Microgrid cores to intercommunicate efficiently allows for the Microgrid to perform better as a group of core than that of the Opteron.

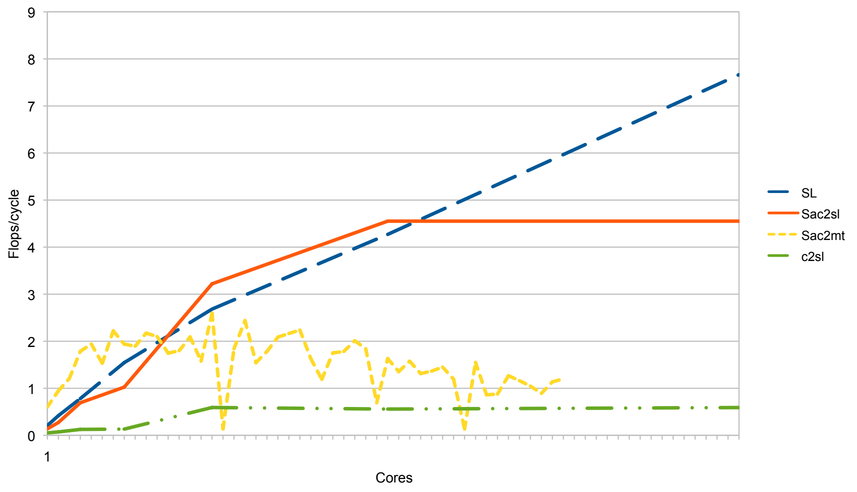


Figure 34: Simulated and non-simulated image smoothing measurements of FLOPS/Cycle for Core counts ranging from 1 to 64.

Figure 35 shows speedup against sequential runs for core counts from 1 to 64. The graph shows how the SL and SAC simulated experiments continue to speed up up to 64 cores.

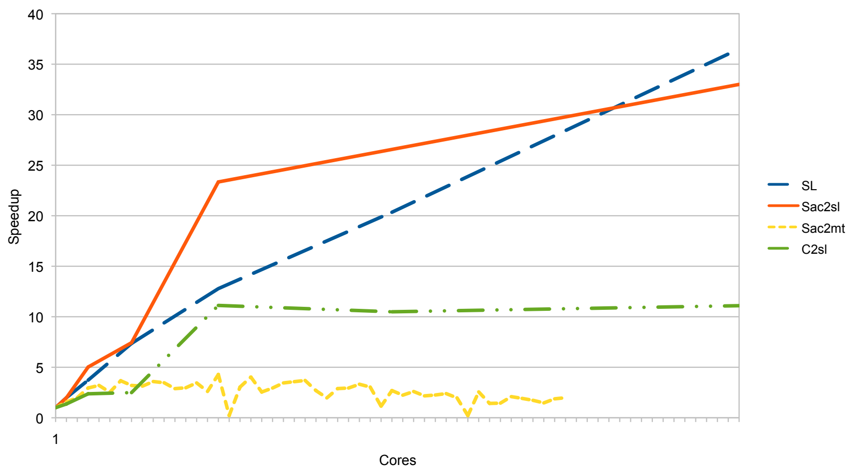


Figure 35: Simulated Image smoothing measurements showing run-times for 1 core divided by the simulated runtime for increasing numbers of cores ranging from 1 to 64.

The graph in Figure 36 shows that in SAC2C computing the result in canonical order performs better than naive ordering. The use of canonical order allows the cache lines of the result to be populated during one visit to the cache. By populating the cache lines in one go there is less demand placed on the external memory interface.

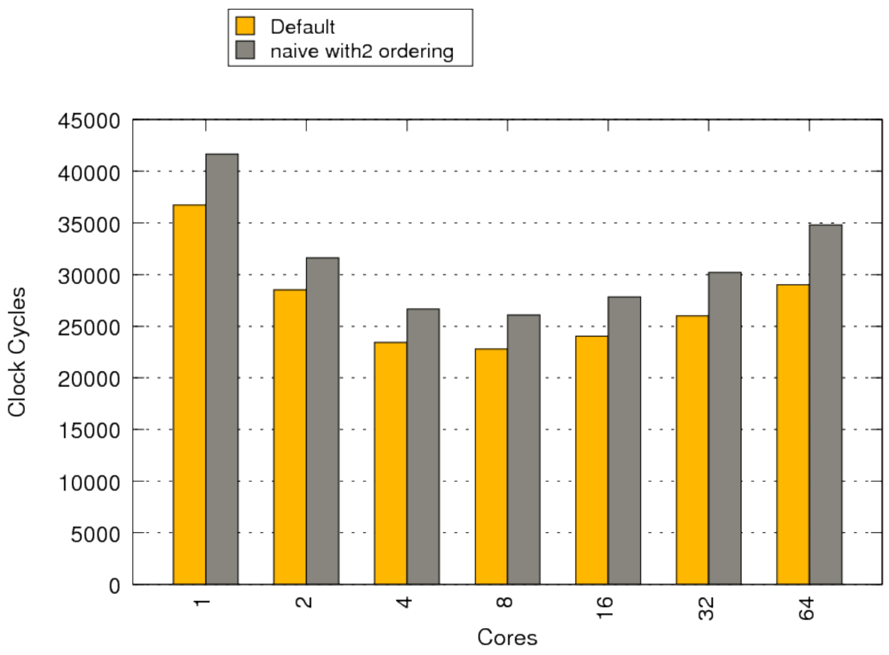


Figure 36: Simulated runtimes in clock cycles for SAC image smoothing implementations for naive and canonical ordering of array elements.

Figure 37 shows the double-precision floating point operations per cycle of increasing problem sizes of the SAC implementation run on a 48-core Opteron machine. Note that on this machine 100 is too low a problem size to achieve anything in excess of three floating point operations per cycle.

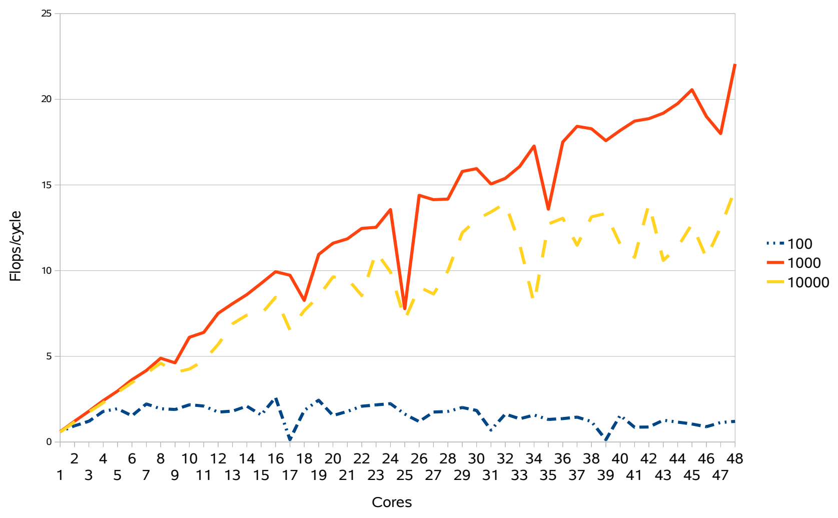


Figure 37: Double precision floating Point operations of the Image Smoothing code on the 48-core Opteron machine.

The latest graphs and results for experiments in this section can be found in Unibench at https://unibench.apple-core.info/?page=benchmark_suites&filterId=155

4.5 Cellular Automata

Many computational models representing physics simulations can be expressed as cellular automata. Cellular automata are widely studied and expose parallelism in models. These models [Gar70] describe what looks like an architecture in itself and whilst these models can trivially be simulated on modern architectures the challenge still exists to do this efficiently. An efficient and generic cellular automata implementation, once written allows further cellular automata based problems to be simulated without any further implementation effort. For this reason that is exactly what we experiment with here.

Here cellular automata simulators are implemented in SAC and SL. Each implementation accepts an initial automata state and a number of steps to simulate over and advances the cellular automata the specified number of steps. Only the computational advancement of the cellular automaton is measured.

Figure 38 shows the simulated experiment run on the SL implementation.

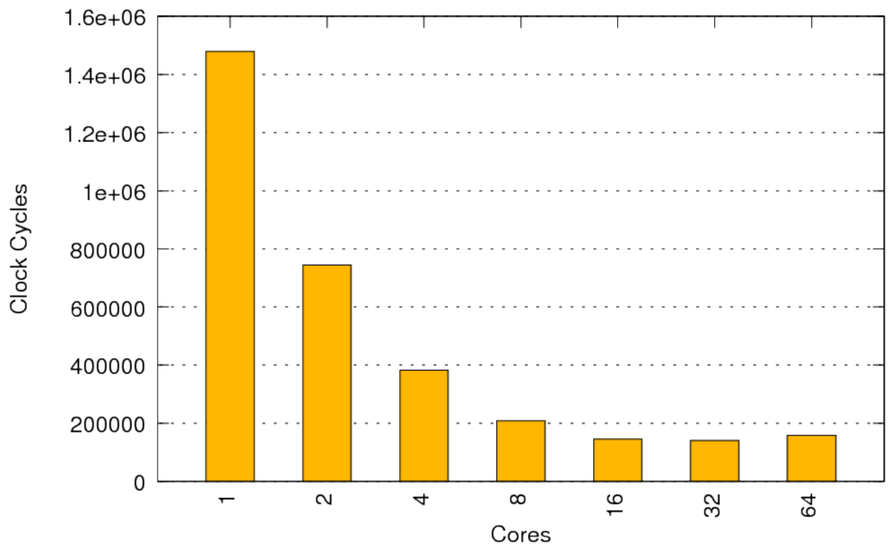


Figure 38: Simulated results for the SL implementation of the Cellular Automata benchmark on the 256 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=597)

Figure 39 shows the simulated experiment run on the SAC implementation.

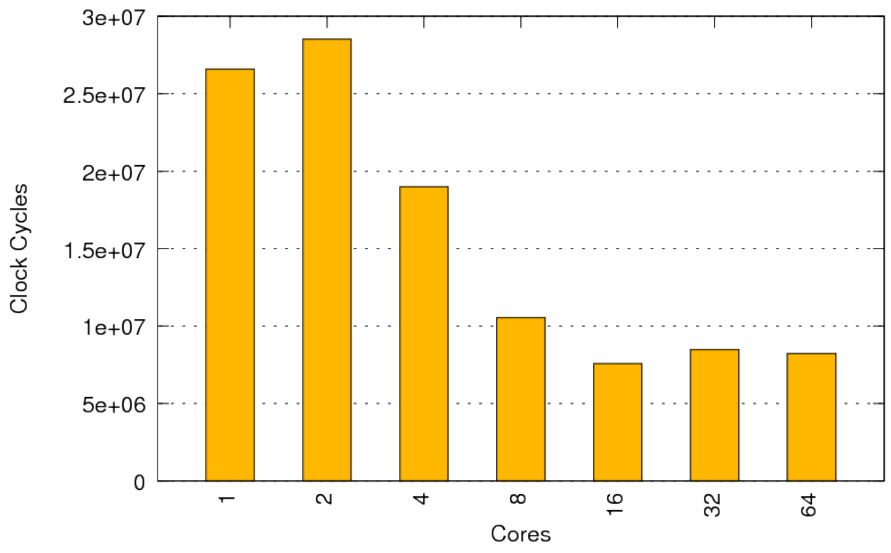


Figure 39: Simulated results for the SAC implementation of the Cellular Automata benchmark on the 256 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=718)

The latest graphs and results for experiments in this section can be found in Unibench at https://unibench.apple-core.info/?page=benchmark_suites&filterId=163

4.6 GEMM

Here we compute the equation

$$\alpha.AB + \beta.C$$

where A, B and C are matrices and α and β are scalers. This allows us to demonstrate that our Matrix multiply implementations works scales well when used in mathematical expressions and isn't just hand tuned to work in isolation. These operation is part of BLAS [Rep].

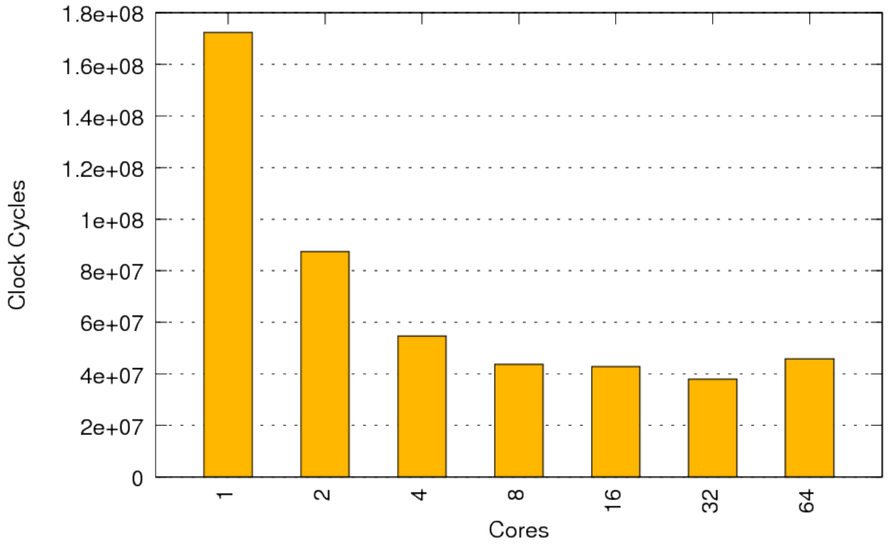


Figure 40: Simulated results for the SAC implementation of the Gauss Elimination benchmark on the 128 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=677)

Figure 40 shows our SAC implementation of this benchmark computing on arrays of size 64 by 64. The speed-ups observable with vanilla matrix multiplication still exist in this setting. For a comprehensive analysis of matrix multiplication see Section 4.8.

4.7 Matrix Inversion

This problem is taken from the MTI Radar applications mentioned in Apple-Core Deliverable 2.2. The MTI radar application [PHG+09] is a signal processing application from the embedded and real-time systems domain. It is under development by Thales Research and it has become a joint project of the aforementioned institution and the University of Hertfordshire. High throughput and low latency are the natural requirements for this application, in which adaptive filter algorithms are applied to incoming radar burst echoes.

The main goal of this project is to re-design existing low-level but high-performance C implementations of signal processing functions and required algorithms in a high level language without sacrificing runtime performance.

This project demonstrates the suitability of SaC as is an implementation language and the Microgrid architecture as execution platform for applications of this industrial domain. For this reason it is imperative to include this application in our pool of chosen applications and benchmarks.

One key part of the code was the matrix inversion algorithm. The experiments here focus solely on this algorithm which was a key computational kernel that needed speeding up to successfully accelerate the MTI Radar application.

Figure 41 shows the performance on the three main memory architectures of the project. Scaling is achieved regardless of the memory architecture but whilst coma and the random banked memory (rmb) have similar performance mlcoma256_4chan heads far greater performance. This backs up a general result from Apple-Core which is that whilst the memory access patterns are important factors on performance in legacy systems on Microgrid systems this situation is even more common.

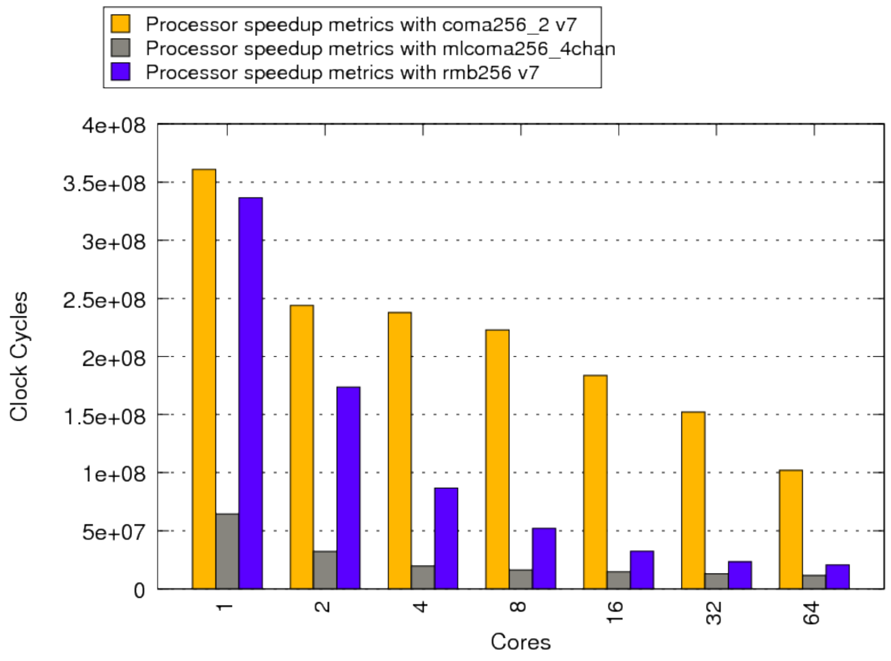


Figure 41: This graph shows the performance of the three main memory architectures used in this project.

Figure 42 shows a SaC implementation of the matrix inversion algorithm on a matrix of doubles containing 10000 elements.

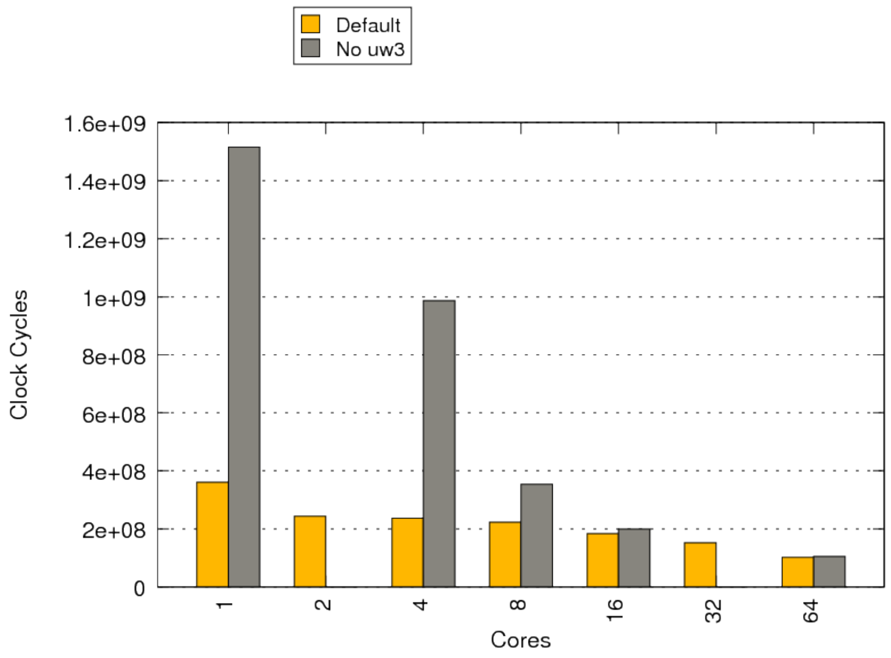


Figure 42: Simulated runs of the matrix inversion algorithm written in SAC applied to a matrix of 100000 elements. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=585)

In Figure 42 we see that when the SAC2C optimisation 'Unroll With3' is disabled the run time increases significantly. The uw3 optimisation unrolls small with3 loop partitions that if not removed would result in small creates. Small creates offer very little resources but use up thread and family table entries. The entries freed by avoiding these small creates can be used in larger creates that expose more concurrency to the architecture. As well as offering reduced use of resources unrolling tiny nested array operations allows for the small overhead of the create to also be removed.

The latest graphs and results for experiments in this section can be found in Unibench at https://unibench.apple-core.info/?page=benchmark_suites&filterId=51

4.8 Matrix Multiplication

Matrix Multiplication is defined as

$$R_{[i,j]} = \sum_k A_{[i,k]} B_{[k,j]}$$

and is commonly defined in a vast number of benchmark suites including BLAS [Rep] and the high-performance challenge benchmark suite [LBD+06]. As a key BLAS subroutine with performance on legacy hardware affected by the memory access pattern of the algorithm matrix multiplication was always a key APPLE-CORE benchmark. Algorithmic performance of matrix multiplication is well researched [BBRR01] and commonly relevant in HPC code.

We want to demonstrate both good scaling on increasing numbers of cores and that the overall performance on the Microgrid architecture shows promise.

Within APPLE-CORE, experiments were run with matrix multiplication simlulated on hardware on all three, C, SAC and SL toolchains and in addition using SAC on legacy platforms. Section 4.8.1 below first gives SAC results on legacy hardware and Section 4.8.2 then documents experiments to reproduce these results with microthreads on simulated hardware.

4.8.1 Legacy Hardware Results

The following experiments were run on an Intel Xeon E5540 2.53 GHz, 2x Quad Core with 24GB RAM. In each experiment ten matrix multiplications of arrays of size 1024 by 1024 were measured.

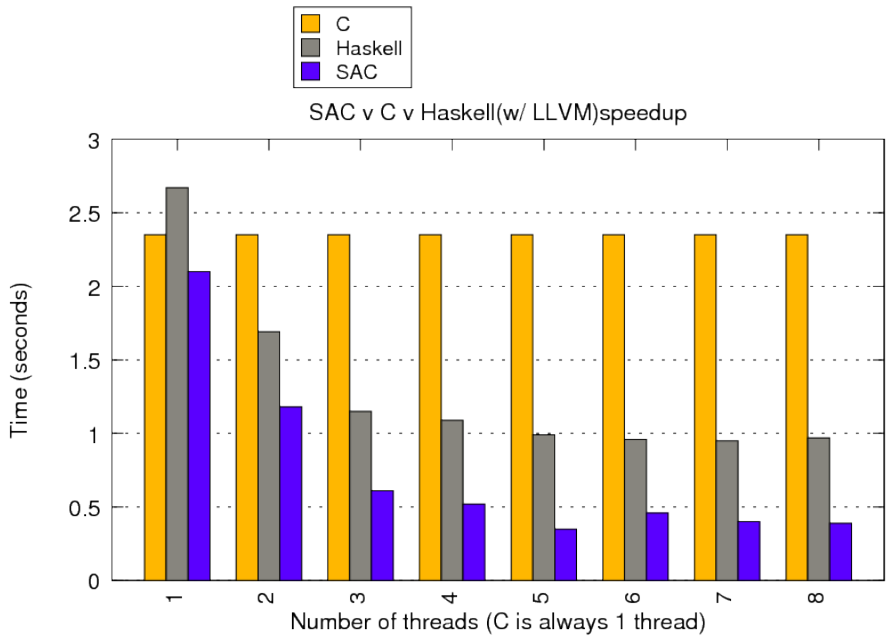


Figure 43: Matrix multiplication runtime on legacy hardware (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=411)

Figure 43 shows the SAC matrix multiplication implementation compared against a naive matrix multiplication implementation in C and Data Parallel Haskell. The results are shown for increasing numbers of cores where for SAC and Haskell an auto-parallelising compiler was used. SAC2C produces more efficient code that the C compiler even on a single core whereas the Haskell code is marginally slower but still competitive. Both Haskell and SAC become noticeably faster than the C implementation immediately as more threads are used for the computation. With the heavyweight PThreads in use and the 1024 by 1024 arrays they were able to scale well up to five threads.

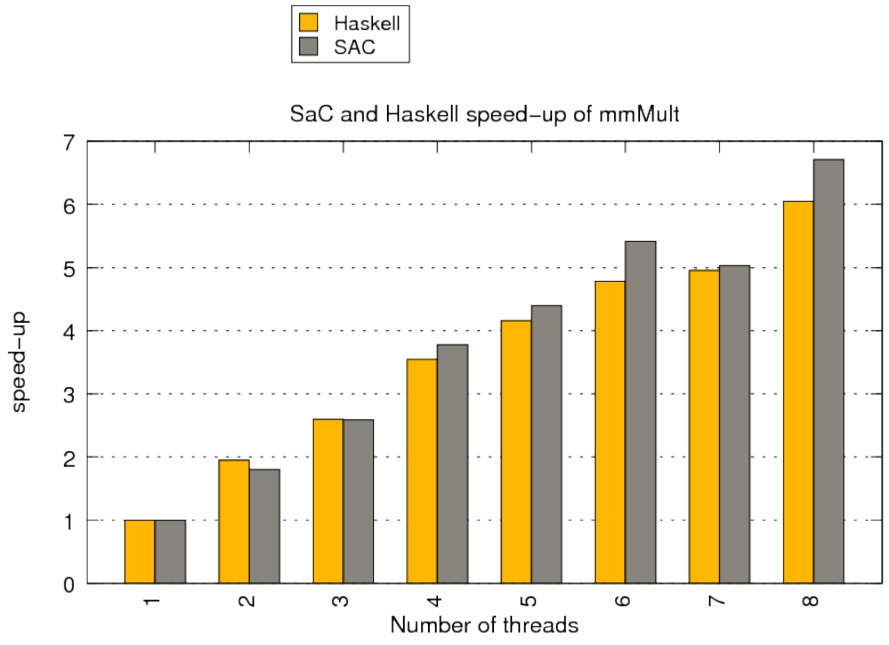


Figure 44: Matrix multiplication speedups on legacy hardware (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=501)

Figure 44 shows the speedup of the SAC and Haskell code for larger arrays of size 5120 by 5120. Note that for these larger arrays, as more threads are provided the runtime continues to speedup for longer. This inability to scale on small numbers of cores for small problem sizes is a problem that the Microgrid can solve.

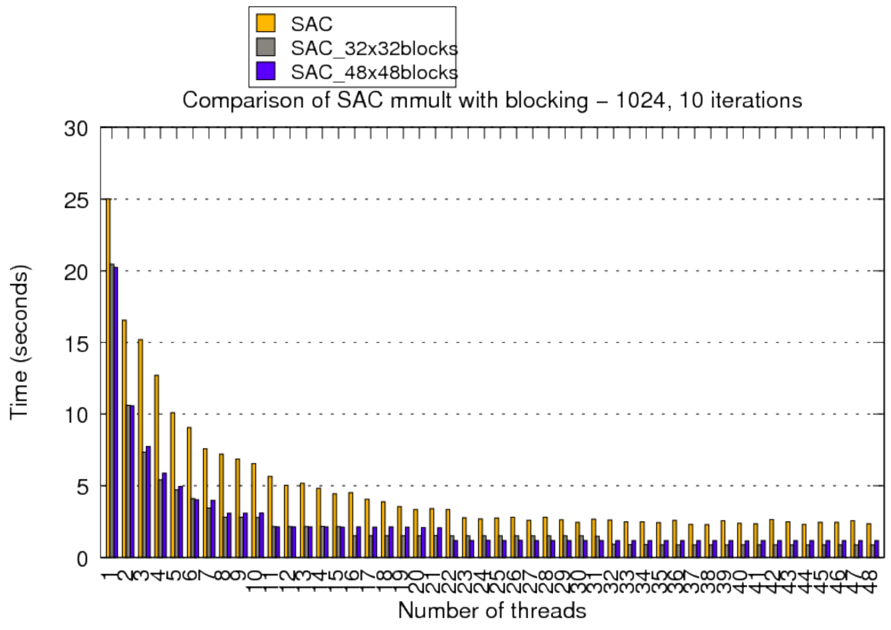


Figure 45: Matrix multiplication runtime on legacy hardware with blocking (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=408)

Finally Figure 45 shows SAC matrix multiplication measurements on a 48-core 2.2GHz AMD Opteron machine with 256MB RAM. Here for arrays of size 1024 by 1024 the former SAC implementation is compared to implementations where blocking is used. In each case the block size is explicitly given to SAC and the blocking algorithm is auto-generated by SAC2C. Blocking improves sequential runtime significantly, as was hoped. Surprisingly, the extra complexity of the blocking algorithm and the way it subdivides arrays does not appear to inhibit the scaling in any way. The fastest runtimes are achieved with blocking.

The latest graphs and results for experiments in this section can be found in Unibench at
https://unibench.apple-core.info/?page=benchmark_suites&filterId=132

4.8.2 Simulated results

Three separate implementations of matrix multiplication algorithms were implemented for all three major Apple-Core toolchains in C, SL and SaC. Measurements show three multiplications of arrays of 30x30 elements on the three toolchains. The runtimes are measured in clock cycles and show only times taken for the multiplications themselves.

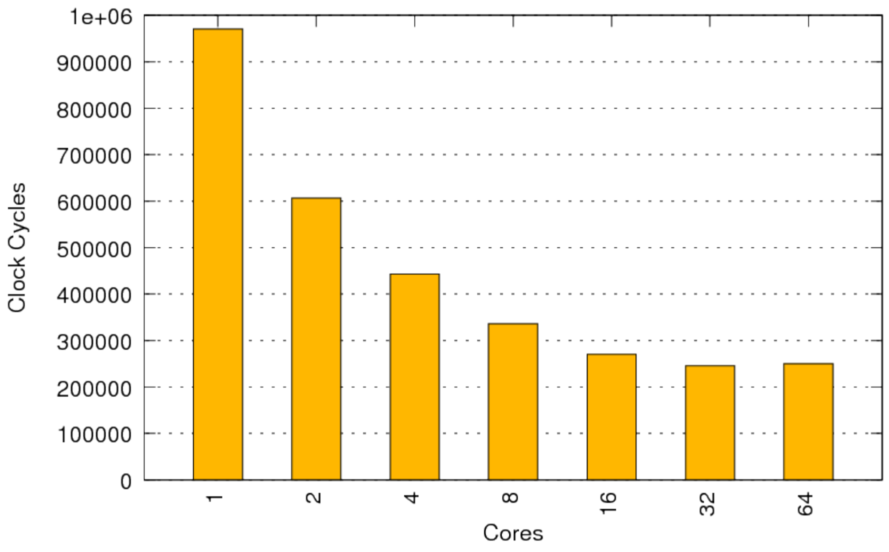


Figure 46: Simulated results for the C implementation of the Matrix Muliplication benchmark on the 256 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=623)

Figure 46 first shows the simulated results for the Apple-Core C compiler. Pleasingly, auto-parallelisation to a significant scale was achieved even with legacy C code.

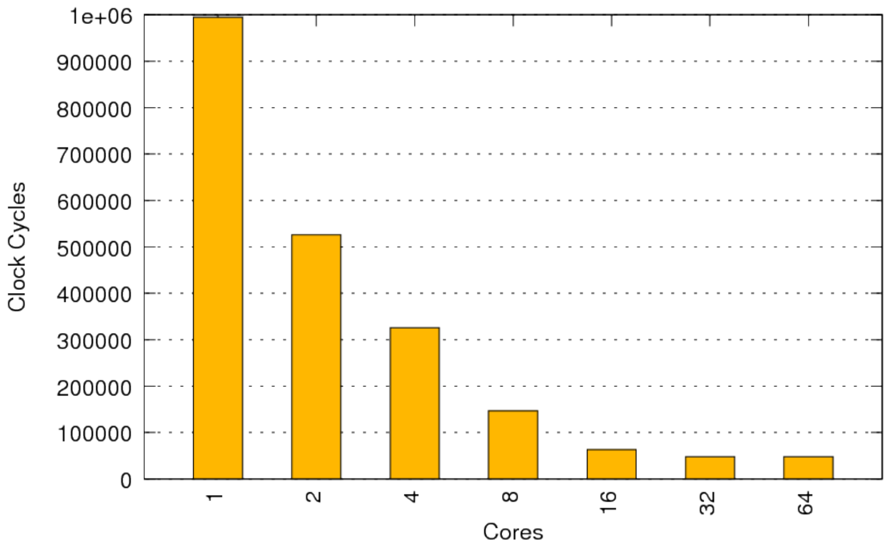


Figure 47: Simulated results for the SL implementation of the Matrix multiplication benchmark on the 256 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=610)

In Figure 47 above the same experiment on the hand-coded SL implementation of the algorithm is shown. Here improved scaling and faster runtimes are demonstrated.

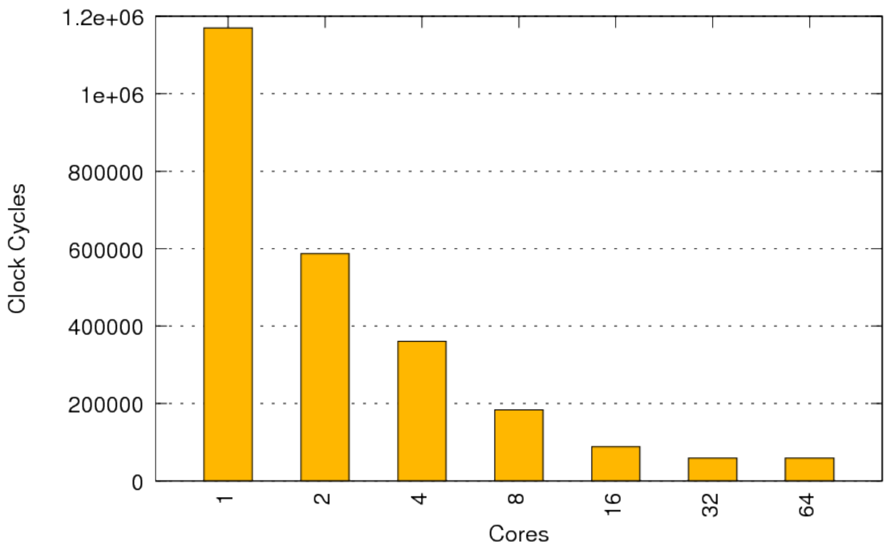


Figure 48: Simulated results for the SAC implementation of the Matrix multiplication benchmark on the 256 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=608)

Finally Figure 48 shows the same experiment run with the SAC implementation of matrix multiplication. On a single core the SAC implementation shows a small amount of overhead against the hand-coded SL implementation. This is to be expected and is small enough to be acceptable before scaling on multiple cores is taken into account.

As the matrices being multiplied are 30 by 30 elements in size sizable speedups can be introduced as the number of cores is raised 30 cores (or 32 on the graph). The rate of this scaling reduces after 30 cores but continues for another 30 cores up to 60 cores. This pattern exists due to the distribution of sets of 30 threads for the inner matrix dimension spawning from each outer dimension's thread. More information on the distrubution scheme and this effect in general can be found in APPLE-CORE Deliverable 4.1. The way in which SAC makes use of nested parallelisem where, each dimension is mapped into a family of threads, performs comparibly with the hand chosen distribution in SL.

To generalise the above, efficient scaling can be achieved up to $2^{\lceil log_2(max(30,30))\rceil} = 32$ cores. Note that in our simulated example relitively small matrices were used but as these are scaled up to large arrays the potential for scaling effiently with larger arrays of cores increases substantially.

The C implementation in Figure 46 actually marginally improves upon the runtime of the hand-coded SL on a single core. All three implementations scale as the number of cores increases. They scale up to 32 cores even though the problem size is only a 30 by 30 matrix. The SAC and hand-coded SL implementations initially scale linearly whereas the C implementation initially scales close to linearly. All runs demonstrate smooth scaling.

The latest graphs and results for experiments in this section can be found in Unibench at https://unibench.apple-core.info/?page=benchmark_suites&filterId=64

4.8.3 Comparisons of legacy vs Microgrid

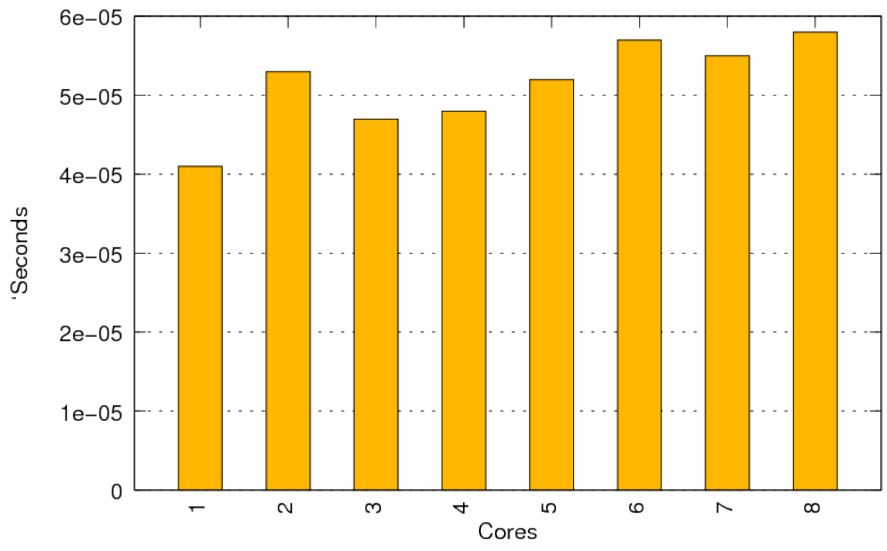


Figure 49: Scaled runtimes for 3 matrix multiplications of 30 by 30 matrices compiled from SAC using SAC2C's PThread backend. The original number of multiplications before scaling was 30,000,000. These experiments were run on an 8 core 2.2Ghz Intel Xeon machine with 24 Gb of RAM. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=697)

Figure 49 shows scaled runtimes for 3 matrix multiplications of 30 by 30 matrices compiled from SAC using SAC2C's PThread backend. For matricies this size runs are too fast for accurate measurements and so the graph shows scaled results from 30 million matrix multiplications down to three for ease of comparisson with the simulated results. As can be seen for arrays of this size no significant speedup can be seen on the PThread implementation. This demonstrates that fine grained parallelism of matrix multiplication computations is now possible with microthreads.

Here the double precission floating point operations per cycle are calculated. Some implementations need extra floating point operations and use more floating point operations than are technically required in the algorithm. A contrived algorithm using far in excess of the number of required operations could artificially achieve higher floating point operations per cycle despite being inefficient. For this reason *all figures are based on the minimum floating point operations required for the calculation* and not on the actual number of operations used. Use of this figure puts the simulated parallel Microgrid experiments at a dissadvantage when large numbers of microthreads are used but we believe it makes the comparissons fairer.

In the case of matrix multiplications the number or arithmetic operations is

$$N.N(N+N-1)$$

or more simply

$$2N^3 - N^2$$

where N is the size of the square matricies being multiplied.

Table 1 shows the double precission floating point operations per second for the above experiment.

Table 1: Double precision floating point operations per cycle for the experiment in Figure 49.

1	0.000100
2	0.000114
3	0.000092
4	0.000089
5	0.000096
6	0.000108
7	0.000103
8	0.000102

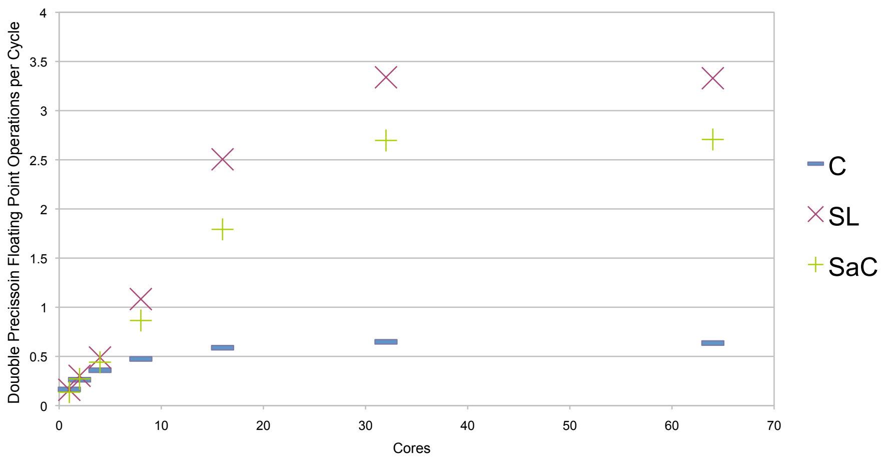


Figure 50: Double floating point operations per cycle for the simulated Micrgrid experiments in Section 4.8.2.

Figure 50 shows the double floating point operations on doubles per cycle for the simulated experiments with C, SaC and SL in Section 4.8.2. The plots for C and SL look similar with SaC performing at a rate close to the ideal set by hand-coded SL. The C code, which is tougher to autoparallelise, still shows non-trivial gains. Comparisson between Figure 50 and Table 1 allow direct comparissons between the same experiments in on legacy systems and the Microgrid. For these small experiments the double precission floating point operations per second on the Microgrid are a lot higher demonstrating that in addition to scaling the computational performance is improved on the Microgrid.

With such small datasets no significant performance gain or scaling is noticable on legacy hardware. Larger datasets are infeasible for simulation but can still be run on legacy hardware: Figure 51 below shows the double precission floating point operations per cycle for the larger experiments in Section 4.8.1. Here we can see that for the considerably larger experiment, legacy hardware can finally outperform the simulated Microgrid results on much small problem sizes. Note that cores in the legacy setting are far more complex and expensive than a core on the Microgrid.

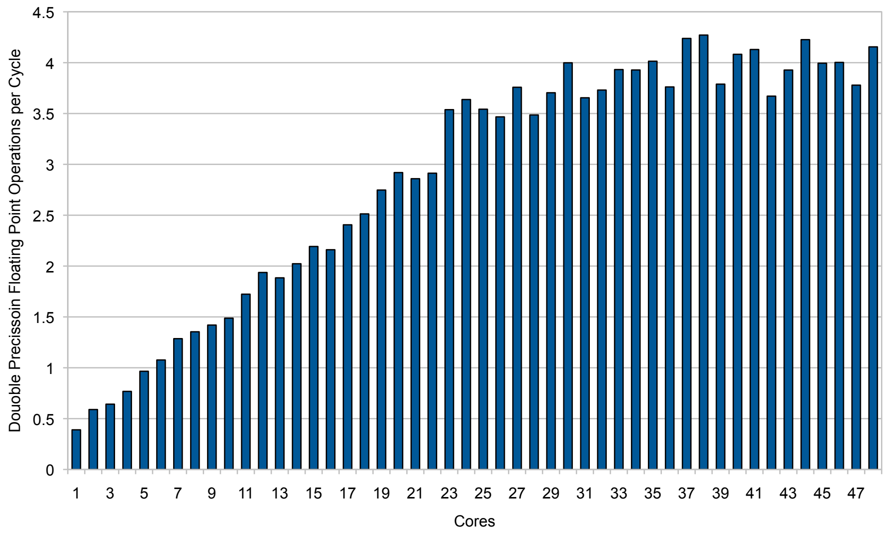


Figure 51: Double-floating point operations per cycle for the larger experiments on legacy code in Section 4.8.1

We considered matrix multiplication an essential benchmark due to its importance in industrial applications and its inclusion in many benchmark suites including BLAS. Parallel speed-ups and competative performace was initially demonstrated as achievable on the Microgrid architcture though hand-coded SL examples. Competative speedups and performance achievements were independently demonstrated on legacy hardware using the high-level SaC language where a high-level specifcation for matrix multiplication was compiled to efficient C code. Fortunately with Apple-Core we were able to use the new toolchain to compile the exact same SaC code to SL and demonstrate that this compiler-produced code can compete with hand-coded SL. Finally C code was shown to auto-parallelisse this essential benchmark for demonstrating the potential of the architecture.

4.9 Fibonacci

The fibonacci series is the series starting $0, 1, \ldots$ [1] where every following number at position n is defined as $\text{fib}(n-1) + \text{fib}(n-1)$. The series is well known in computer science. We are interested in using this simple series to explore the potentials of functional currency on the Microgrid in an extreme situation. We use a simple recursive algorithm which is intentionally not the most efficient.

```
function fib( n):
  if n == 1:
    return 0
  if n == 2:
    return 1
  else:
    return fib(n-1) + fib(n-2)
```

This function is conceptually suitable for parellelising since in the common else block two functions are called recursively and they could both be executed in parallel. If functional parallelism were used to achieve this effect then each thread would either simply return a value or just spawn two more threads and perform a simple addition. This is a tiny workload which would usually make parallelisation impracticable. On the microgrid fine grained parallelism is a reality and we wish to see how far we can push this idea. We believe this to be an extreme example to test how far the architecture can be pushed. If we were to simplify it further by removing the addition then we could not practically ensure that we return a correct result.

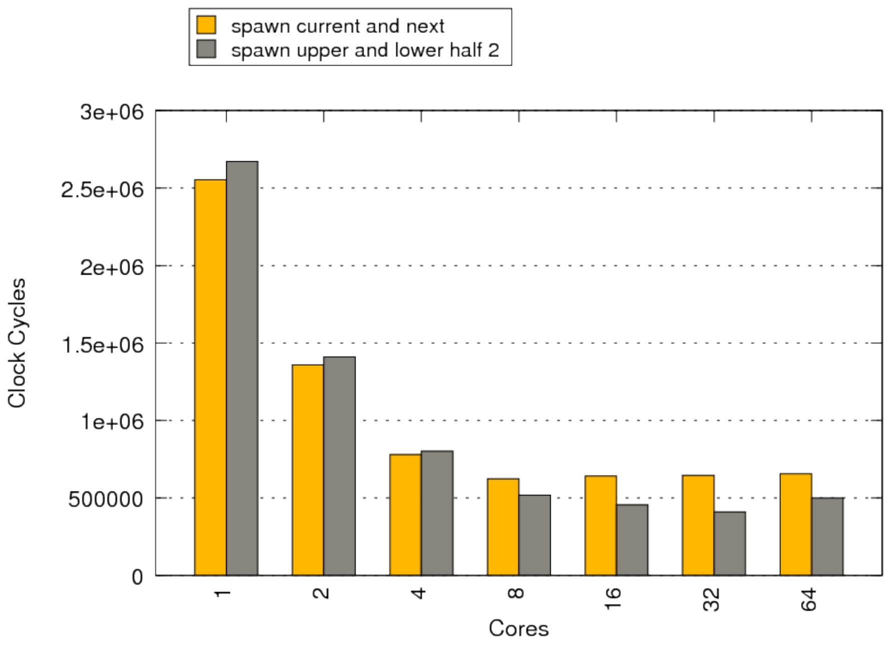


Figure 52: Simulated results for the SAC implementation of the Fibonacci benchmark on the 128 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=675)

Figure 52 shows the computation times to caluculate the 20th fibonacci number using the naive algorithm above and spawning a new thread on each recursive call. In the graph two distribution schemes are compared. In the first 'spawn current and next,' the first spawn always places the thread on the cuurent core and the second spawn always places the thread on the next core in the ring. In the second, 'spawn upper and ower half,' the first spawn places the thread on the next core in the ring and the second core places

[1]Some definitions show the series starting $1, 1, \ldots$

For this example where each spawned function call has very little actual computation it appears that the scheme for distributing threads onto the cores makes little difference. This is unsuprising since these threads require little memory access. An interesting result was that even in this extreme scenario with very little actual compuation to distibute scaling is possbile. Whilst Microthreads all very fine grain parallelism we never expected to achieve gain almost linear scaling up tp four cores as was achieved here. To push functional concurency further and explore its applicability to application benchmarking we go on in the next section to experiment with a sorting algorithm.

4.10 Quicksort

Quicksort is a famous and extremely well known and sucessful sorting algorithm developed by Tony Hoare[Hoa61]. On average it makes $\bigcirc(n \log n)$ comparissons to sort a list of length n. It's success and known efficiency alone makes it of interest to the Apple-Core project. It's algorithm involves picking a 'pivot' from the list and creating two sub lists to recursively sort. These are the list of elements greater than the pivot and the list of elements less than the pivot. These two recursions are obviously candidates for functional concurrency and it is this idea that we have experimented with for this benchmark.

The pseudo c/SaC-like code below illustrates the basic idea. When given an empty list the function returns an empty list. Otherwse a pivot is first selected. Then two lists of elements in the first list greater and less than the pivot are generated. Two recursive calls to quicksort are made on these new lists and the returned value is the concatenation of these lists before and after the pivot. The ++ symbol denotes concatenation.

```
function quicksort( list):
  if empty( list):
    return []
  else:
    pivot = list[0]
    filterL, filterR = filter( list, pivot);
    qsortL = spawn quicksort( filterL);
    qsortR = spawn quicksort( filterR);
    return qsortL ++ [pivot] ++ qsortR
```

Naively, one might think that this obvious opportunity for parallelism naturally means that speedup is inevitable. Unfortunately there are two realities hindering this idea. Firstly, the binary tree that might be imagined from the two recursive and parallelisable calls in each recursion is not necessarily a balanced tree. Even with random lists incredibly large list sizes would be required to make tree noticeably balanced on average. Secondly, the filtering for the elements greater and less than a pivot and creation of two sublists from using the results from this filter is expensive. This happens at each call level before more calls are possible. it is possible to split this into two filters in parallel but no more and this inhibits opportunities for parallelism early on in the runtime. What is interesting is to discovered how much this hinders the potential for parallel speedup.

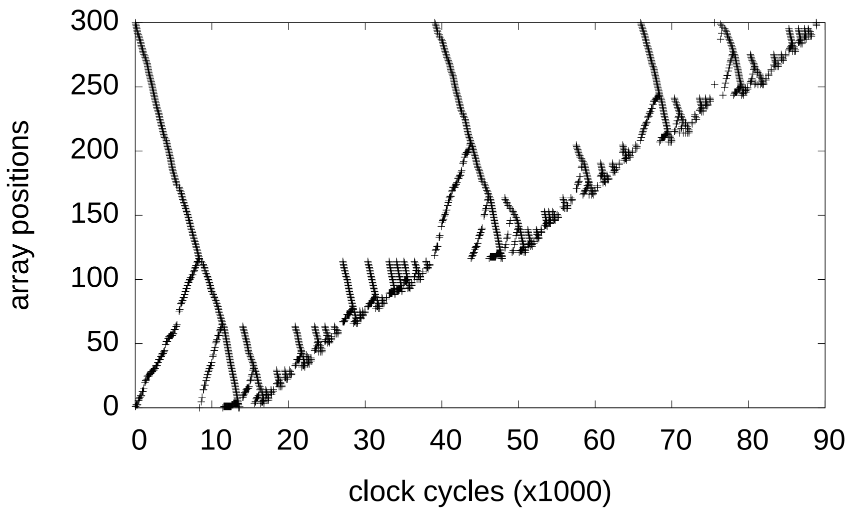


Figure 53: Memory accesses to array elements over time during an hand-coded SL, sequential quicksort run.

A program running quicksort sequentually and in-place on an array was written in SL. Figure 53 shows memory accesses to this array over time for this program. For the first 9,000 cycles memory accesses run from the beginning and end of the array swapping elements larger than the pivot until both sides meet creating two partitions called recursively. They run from 9,000 to 39,000 and 39,000 to 90,000 cycles respectively. Note that the numbers of elements greater and less than the pivot is unbalanced.

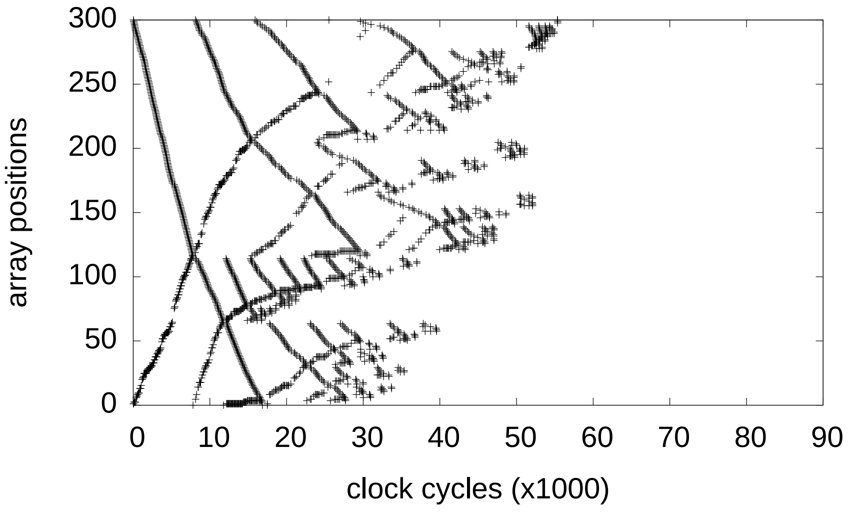


Figure 54: Memory accesses to array elements over time during an hand-coded SL, parallel quicksort run.

Figure 53 shows the same just now sorted in parallel. The first 9,000 cycles cannot be sped up but from this point both recursive calls to the list greater than and less than the pivot can be run in parallel.

Initial attempts to achieve efficient functional concurrency passed the problem of distributing the work among the cores to the low level SL language. The allocation scheme was the following:

1. The allocation message goes from the creating core to the first core in the target place, using the delegation network

2. From that point, the allocation message hops from core to core within the place, using the link network, collecting the current load on each core.

3. When it reaches the last core, the core with the least load is selected, and the allocation message hops back to that core via the delegation network

4. The allocation happens on that core (that core only: no distribution of a balanced family), then the result is sent back to the creating thread

Later the scheme was modified such that if at step 2 any core that has less than N families allocated, where N is the ‘balance threshold’ (N defaults to 1), this core is immediately selected for allocation and step 4 occurs directly. This avoids scanning the entire place

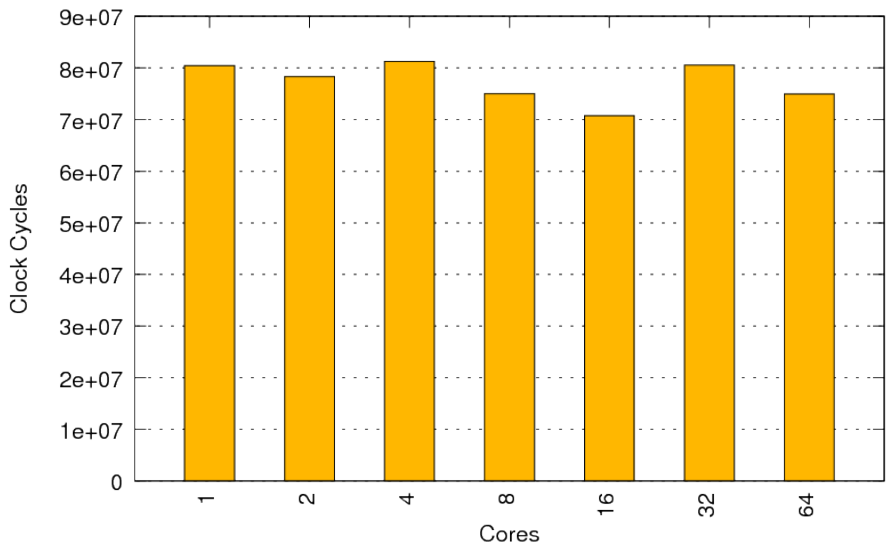


Figure 55: Simulated results for the SAC implementation of the Quicksort benchmark on the 128 architecture.. This experiment uses the 'balanced' strategy. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=732)

The original idea was that the application programmer should not need to be concerned with this. when originally pursuing this strategy only very small speedups where achieved. The strategy was then changed to allow the application programmer to specify a distribution scheme in his program. This allowed us to produce the results presented below. Figure 55 shows the the result of the first attempts using this approach. With this initial attempt the scaling was poor.

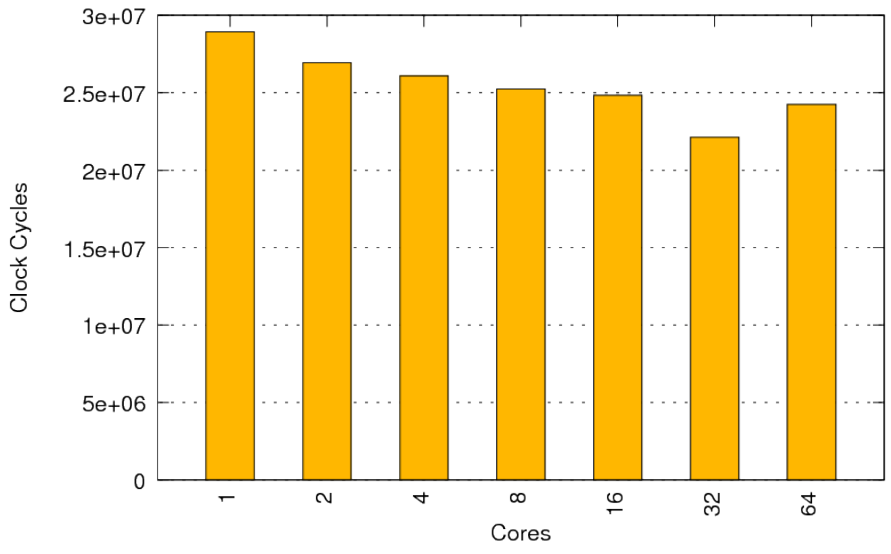


Figure 56: Simulated results for the SAC implementation of the Quicksort benchmark on the 128 architecture. This experiment uses the 'balanced' strategy and switched to a fast sequential version on lists of 10 elements or less. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=728)

Figure 56 shows the same experiment where now the code switches to a fast sequential algorithm on lists of less than ten elements to attempt to speedup the code. This drastically improves the overall runtimes but does not yet yield the desired speedups.

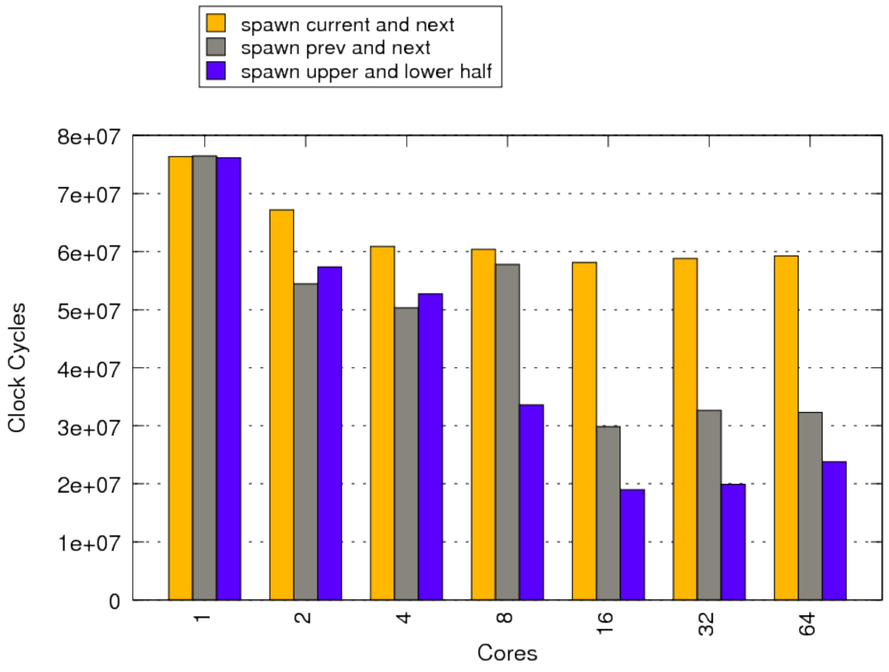


Figure 57: Simulated results for the SAC implementation of the Quicksort benchmark on the 128 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=682)

Figure 57 shows our SAC recursive Quicksort implementation run with three different distribution strategies. Every spawn call in SAC can be passed a parameter to influnce where the spawned thread is placed. The following options are available:

- *NEXT_N_GCORE(N)* Spawn on the core N places forward along the ring. N may be zero.
- *PREV_N_GCORE(N)* Spawn on the core N places backwards along the ring. N may be zero.
- *NEXT_HALF_GCORE* Spawn on the core half way around the ring from the current core.

For the two recursive functional calls in the recursive function the calls are distributed using the following strategies which make use of the options above to specify the placement of both recursive function calls.

- *Spawn Current and next* Spawn on the current and next core in the ring.
- *Spawn prev and next* Spawn on the previous and next core in the ring
- *Spawn upper and lower half* Spawn on the next core on the ring and the core half way around the ring

The current and next strategy has the worst scaling which shouldn't be surprising since it tends not to distribute around the cores very well: the first few cores on the ring tend to host lots of threads whist the later cores are starved of work. We verified this pattern using the debug environment for the simulator. The previous and next strategy initially performs the best but the results are erratic and this strategy does not scale well up to the full 64 cores. The distribution initially spreads out more but quickly the cores around the first core tend to be more overloaded than other cores around the ring. The last strategy distributes on the next core and the core half way around the ring. This simple pattern ensures that right from the beginning the workload starts to spread right around the ring. The initial speedup for this scheme is slower than the previous and next strategy which we put down to the communication costs to distribute right around the ring rather than just to a neighbouring core but this pays of greatly as the number of cores is increased.

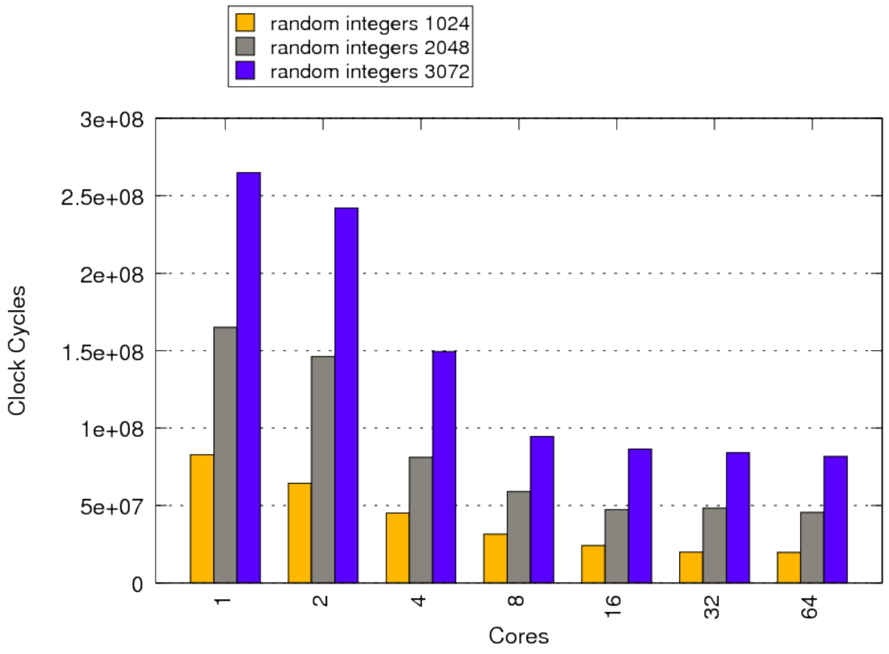


Figure 58: Simulated results for the SaC implementation of the Quicksort benchmark on the 128 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=723)

Figure 58 shows our best strategy (upper and lower half) shown for increasing list sizes of randomly sorted inputs. The input size doesn't appear to have a strong affect on the scaling and runtimes appear to consistently scale linearly with the problem size.

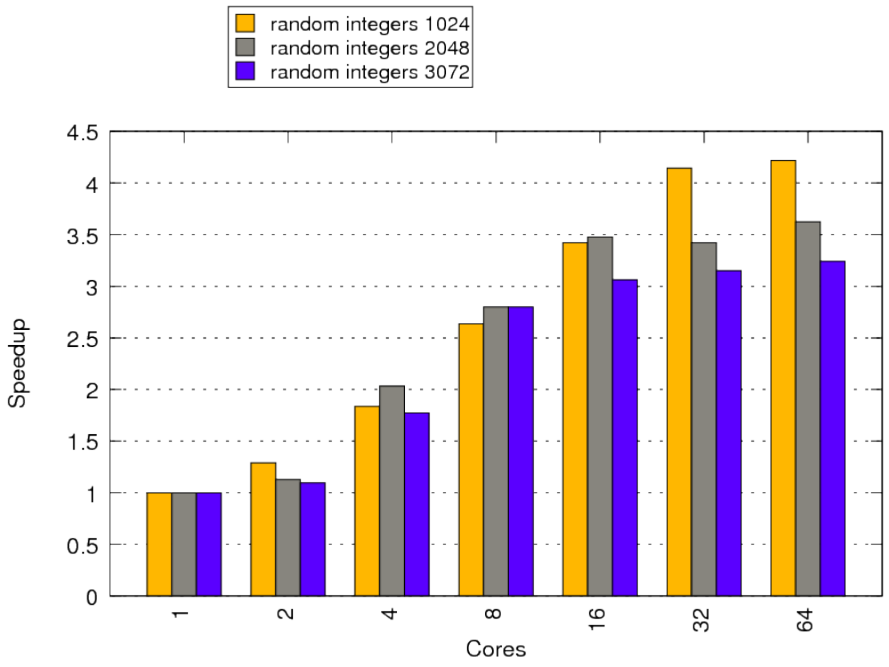


Figure 59: Simulated results for the SaC implementation of the Quicksort benchmark on the 128 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=724)

Figure 59 shows the speedups for the results in Figure 58. This graph reveals more: as the problem size gets large the speedup reduced. Clearly it can still pay off to limit the amount of functional concurrency and not to through as much at the problem as can be discovered.

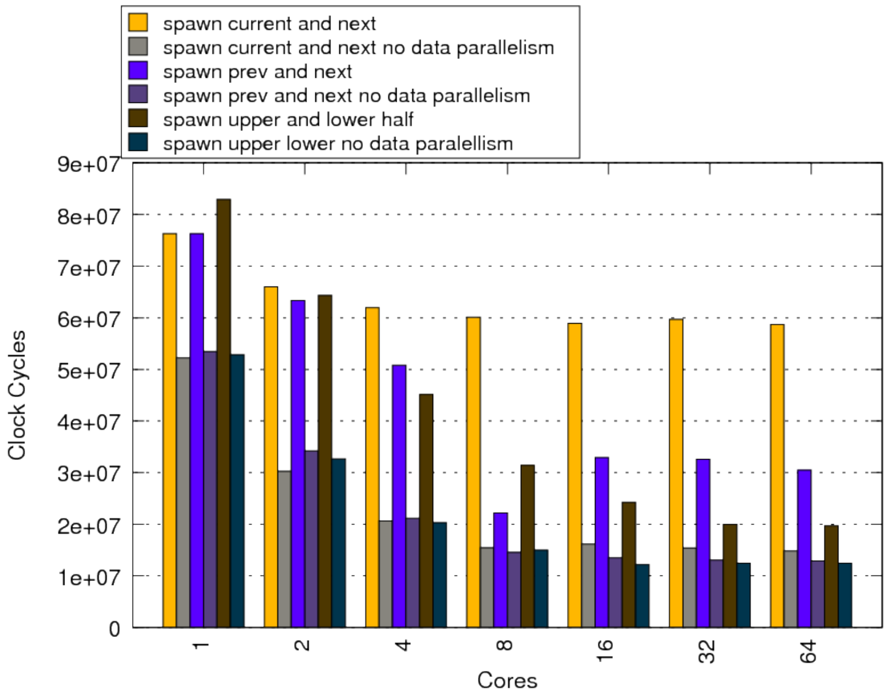


Figure 60: Simulated results for the SAC implementation of the Quicksort benchmark on the 128 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=720)

The concatenation operation in the algorithm can be noticeably expensive. This operation is compiled by SAC2C into data-parallel code. That means that this example contains both data-parallel code and functional concurrency in each run. Figure 60 shows the results from the previous graph (Figure 57) where now for each scheme the runtime with data-parallelism is shown too.

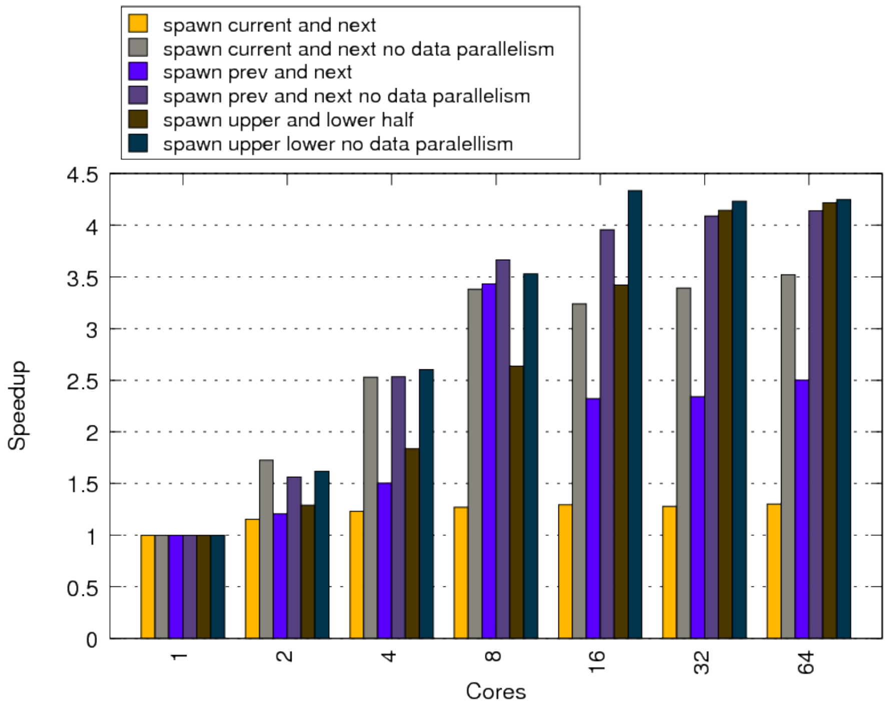


Figure 61: Simulated results for the SAC implementation of the Quicksort benchmark on the 128 architecture. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=721)

Figure 61 shows the same results as speedups. The graphs show that for all distribution strategies removing data-parallelism can help improve performance. This strongly suggests that for now the application programmer should keep control of interleaving data parallelism and functional concurrency.

Finally Figure 62 shows the hybrid quicksort implementation that switches to a fast sequential mode on lists of ten elements or less running with the most sequential thread distribution strategy (upper and lower). This implementation runs twice as fast as the implementation that doesn't switch to the fast sequential mode. It is also a noticable improvement on the results for the balanced scheme.

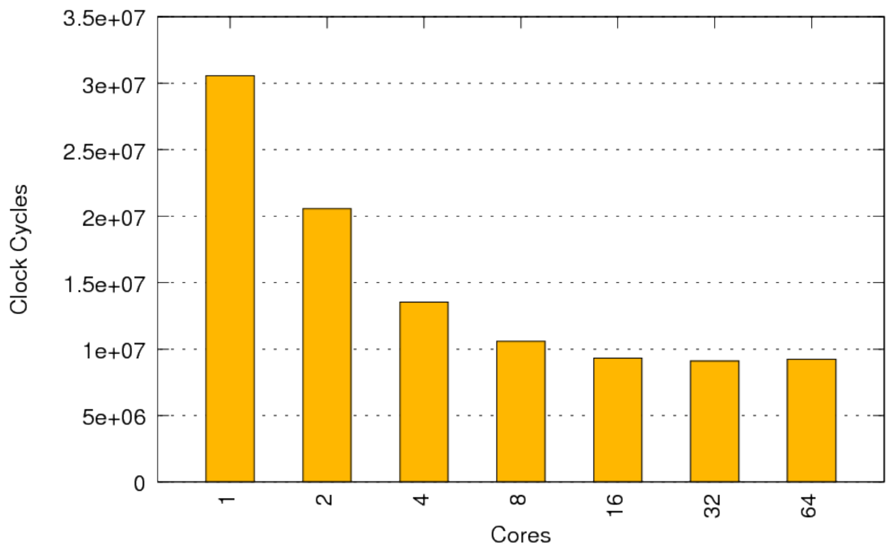


Figure 62: Simulated results for the SaC implementation of the Quicksort benchmark on the 128 architecture. This implementation switches to the fast sequential mode on lists less than or equal to 10 elements and uses the 'upper lower' thread distribution scheme. (Unibench Graph: https://unibench.sac-home.org/?page=showGraph&graph=734)

One obvious finding from these results is that for now it is important that control of thread distribution stays early in the toolchain and can be influenced by the application programmer. For more information on the distribution schemes, their potential and to see a detailed analysis for the a 2D FFT see Apple-Core Deliverable 4.4. In particular this shows that there is still may be better schemes available that yield better distributions and scale to even larger problem sizes.

Section 5 - Conclusion

5.1 Lessons Learnt

Here some of the main lessons learnt based on analysis of the results from runs on the simulated hardware throughout the project are outlined.

5.1.1 Memory Management

Memory access patterns and their respective cache hit rates are important indicators of performance for application writers. Understanding the memory bottleneck is key to program optimisation. The Microgrid does not make this problem go away. Naive use of memory quickly exasperates the problem. The architecture's ability to hide latencies behind active threads are excellent, but they only do work if (i) enough active threads can be created, and (ii) dynamic memory allocations and deallocations can be done effectively and as asynchronous as possible. To achieve acceptable application performance thread local stacks and asynchronous memory managment techniques are now used wherever possible to limit contention between threads.

5.1.2 The importance of Reference counting modes

For large complex applications dynamic memory allocation becomes essential. Modern software heavily relies upon it and SaC makes good us of it in high performance computing. The Microgrid supports running computations with large numbers of threads where it is expected that many of these threads will spend much of their time waiting for memory access. As long as there is at least one thread on each core that isn't blocked accessing memory then the cores can be kept busy. This means however that when creating threads on mass care has to be taken to ensure that the cores don't all need to access the same memory at the same time. Sac2c can optimise code by removing unnecessary copies of arrays and this is commonly essential to performance but should elements of these arrays be shared between multiple threads then these threads may need to communicated via memory to guarantee a thread-safe program execution. We have found, particularly though analysis of the FFT benchmark and of functional concurrency, that the scheme used for reference counting can dramatically affect performance When dynamically allocating memory, reference counting modes become key. Many schemes were implemented and experimented with late in the project and Deliverable 4.4 documents this effort in detail.

We suspect that a deep understanding of reference counting and its effects is key to gaining the best performance from many-core architectures in general, and the Microgrid in particular.

5.1.3 Thread distribution

Early in the project concurrency was managed by the hardware. As the project continued it became increasingly apparent that more decisions on thread creation and allocation had to be made by software early in in the tool-chain. Limited hardware resources have meant that even in a Microthreading architecture, creating new threads at every opportunity is not possible or desirable. This was discovered with data parallelism where fundamental decisions had to be made on which dimensions of arrays to turn to thread functions and on whether to flatten arrays. Functional concurrency brought the same trouble and eventually individual spawned function calls where annotated with distribution strategies in the application code itself. We hope to enable the Sac2c compiler itself to choose between these schemes in the future.

5.2 Future Work

When simulating hardware, actual runtimes are significantly higher than the simulated runtimes. Consequently, benchmarks in Apple-Core were carefully designed to avoid using expensive, unnecessary function calls for printing outputs or analysing results within simulations. Also when working with research and prototype software projects and integrating these code-bases ensuring stability and consistency of interfaces can be a problem. Unibench has helped the project cope with this but the fundamental problems are inevitable. Experiments have generally had to be small to enable us to realistically get results for comparisons. The more complex applications we envisioned to investigate at the outset of the project turned out to be way beyond what could be realistically done on a simulator base. Various techniques including monitoring pipeline efficiencies, operations per cycle and scaling have enabled comparisons but clearly much more can be done once measurements no longer need to be simulated in software. Even FGPA simulations could soon enable far larger problem sizes to be run and for better comparisons that take into account cache effects and memory access patterns into large arrays of data. SAC has already be shown to perform well [KRSS09] on numerical codes on legacy hardware and the results from APPLE-CORE show that there is substantial potential to improve on these results were a Microgrid is available.

Given the complexities of the toolchain, limited resources and need to constantly evolve the code-bases but yet keep the various sub-projects integrated we believe that simply keeping the codebases up-to-date and enabling cross-toolchain comparisons has been an achievement. We were successful in producing code with auto-parallelisation from high-level languages to code that almost identically matched the hand-coded SL examples. For further exploitation it is imperative that we can run larger, more complex examples with some form of hardware acceleration to properly compare with the state-of-the-art. The results of the simulated runs have shown that this is a worthy thing to do and that the Microthreading paradigm has great potential.

Our work on Quicksort and FFT in this report and APPLE-CORE Deliverable 4.4 have made clear that although we have found thread distribution schemes for functional concurrency there that have yielded good performance there is still scope for more investigation. With recent improvements in reference counting techniques (again documented in Deliverable 4.4) it has become clear the data-parallelism and its interactions with functional parallelism are much more intricate than what meets the eye and are definitely worth exploring further.

5.3 Summary

A range of critical benchmarking kernels and applications have been shown to be auto-parallelisable with the SAC2C compiler and the data parallel code that the SAC2C compiler produces has been shown to perform and scale well when simulated on the Microgrid architecture. Moreover functional concurrency is available for additional acceleration even when data parallelism also occurs. Legacy C code can also benefit from the APPLE-CORE many-core technology and in many cases can scale as well as non legacy code. The non-legacy code can however parallelise much code that is known to be extremely difficult to tackle in legacy code.

The Microgrid architecture shows great potential for applications specialised for high performance computing, for embedded computing and for general purpose computing. High-level languages like SAC expose data parallelism on a large scale and tool-chains can make use of the architecture to produce efficient parallel code utilising hardware threads to compute this code. In addition, vast amounts of parallelism can be generated and exploited in legacy code through loop analysis techniques.

References

[ABC$^+$06] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelik. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California at Berkeley, December 2006.

[BBRR01] Olivier Beaumont, Vincent Boudet, Fabrice Rastello, and Yves Robert. Matrix multiplication on heterogeneous platforms. *IEEE Transactions on Parallel and Distributed Systems*, 12:1033–1051, 2001.

[Gar70] Martin Gardner. *Mathematical Games - The fantastic combinations of John Conway's new solitaire game 'life'.* 1970.

[GHS06] Clemens Grelck, Karsten Hinckfuß, and Sven-Bodo Scholz. With-Loop Fusion for Data Locality and Parallelism. In Andrew Butterfield, editor, *Implementation and Application of Functional Languages, 17th International Workshop (IFL'05), Dublin, Ireland, Revised Selected Papers*, volume 4015 of *Lecture Notes in Computer Science*, pages 178–195. Springer-Verlag, Berlin, Heidelberg, New York, 2006.

[GST04] Clemens Grelck, Sven-Bodo Scholz, and Kai Trojahner. With-Loop Scalarization: Merging Nested Array Operations. In Phil Trinder and Greg Michaelson, editors, *Implementation of Functional Languages, 15th International Workshop (IFL'03), Edinburgh, Scotland, UK, Revised Selected Papers*, volume 3145 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 2004.

[Hoa61] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of The ACM*, 4, 1961.

[JLZ09] Chris Jesshope, Mike Lankamp, and Li Zhang. The Implementation of an SVP Many-core Processor and the Evaluation of its Memory Architecture. *ACM SIGARCH Computer Architecture News*, 37(2):38–45, 2009.

[KRSS09] Alexei Kudryavtsev, Daniel Rolls, Sven-Bodo Scholz, and Alex Shafarenko. Numerical simulations of unsteady shock wave interactions using SAC and Fortran-90. In *10th International Conference on Parallel Computing Technologies.* accepted, 2009.

[LBD$^+$06] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The hpc challenge (hpcc) benchmark suite. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 213, New York, NY, USA, 2006. ACM.

[McM86] F. H. McMahon. Livermore fortran kernels: A computer test of the numerical performance range. Technical report, Lawrence Livermore National Lab., CA., 1986.

[nDYRHK06] null Dong Ye, J. Ray, C. Harle, and D. Kaeli. Performance characterization of spec cpu2006 integer benchmarks on x86-64 architecture. *IEEE Workload Characterization Symposium*, 0:120–127, 2006.

[PHG$^+$09] Frank Penczek, Stephan Herhut, Clemens Grelck, Sven-Bodo Scholz, Alex Shafarenko, Rémi Barrere, and Eric Lenormand. Modelling a parallel signal processing application with S-Net and SAC. submitted, under review, April 2009.

[Rep] The Netlib Repository. Basic Linear Algebra Subprograms (BLAS). `http://www.netlib.org/blas/`. website accessed: May 2009.

[RHJS09] Daniel Rolls, Stephan Herhut, Carl Joslin, and Sven-Bodo Scholz. Unibench: The swiss army knife for collaborative, automated benchmarking. In: IFL 09: Draft Proceedings of the 21st Symposium on Implementation and Application of Functional

Languages. SHU-TR-CS-2009-09-1, Seton Hall University, South Orange, NJ, USA., 2009.

[WS93] M. S. Warren and J. K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 12–21, New York, NY, USA, 1993. ACM.