Analysis of Power Management Strategies for a Single-Chip Cloud Computer

Robert Eksten[†], Kameswar Rao Vaddina[*][†], Pasi Liljeberg[†] and Juha Plosila[†]
[*]Turku Center for Computer Science (TUCS), Joukahaisenkatu 3-5, 20520 Turku, Finland
[†]Department of Information Technology, University of Turku, Turku, Finland
Email: {roekst, vadrao, pakrli, juplos}@utu.fi

***Abstract*—The 48-core SCC allows us to test and analyze the efficiency of power management algorithms in a multicore environment. In this paper we explain how we implemented these algorithms on the Intel SCC and how they performed in comparison to each other. Two of the algorithms, Core-Pool and PAST, were introduced in previous research articles and one algorithm, Derivate, was developed in this research project. The benchmark chosen to test these algorithms was a NASA Advanced Supercomputing benchmark that was imported to the platform. We analyzed the algorithms according to the execution time of the benchmark and the average power consumption while the test was running. The algorithms were compared against each other and reference values while having the benchmark running at constant power level. Through this analysis, we were able to determine that Core-Pool gives the best results in power efficiency with PAST close behind with almost as promising results. Derivate on the other hand proved to be still lacking, and in need of redesign in order to provide results that are comparable to the other power management algorithms.**

***Index Terms*—power management algorithms, the single-chip cloud computer, multicore power management, dynamic voltage and frequency scaling.**

I. Introduction

THE market for battery powered consumer electronics is growing rapidly. These appliances have very stringent constraints on the amount of power dissipation of their components. As the complexity and performance of the systems increases by the ever increasing user demand, reducing power dissipation becomes a primary concern of the system designer. Excessive power dissipation leads to increase in on-chip temperatures which increases the cost and noise for complex cooling solutions. Novel low-power design methodologies [1][2] have been suggested for chip-level designs. But more often than not electronic systems are more complex than a single chip. Today's mobile phones and laptop computers often consist of tens or sometimes hundreds of components with each components having a different power dissipation profile. Managing such complex array of components in modern systems need complex power management techniques.

Future complex systems would have multicores at their heart. These multicore systems can be either homogeneous or heterogeneous systems and may use different communication paradigms like Network on a chip. Some simpler methods of fine grain power management techniques for multicore systems have been proposed by researchers from Intel [8][9]. Some of the techniques they suggest include, 1) turning of idle cores to save leakage power, 2) providing each core with two supply voltages and two sleep transistors to select a supply voltage and 3) providing each core with a frequency divider to select the frequency of operation. This greatly simplifies the design and power delivery mechanism. On the other hand any power management in a network is difficult, because power management techniques like clock gating or sleep-transistors may incur wakeup latency thereby impacting the performance of the system [8][9]. Although, in a network on a chip, cores with smaller area give more performance throughput at lower power envelope, they increase the network power consumption. So, a holistic approach is necessary in order to study the system power and performance as the field of power-aware architectural design and optimization becomes important for many-core designers because of the immense and unique challenges involved. In this work we implement various power management techniques to control the power dissipation of Intel's single chip cloud computer.

Previous research done by [6] indicates that power management algorithms can have an impact on the power efficiency of the system which was demonstrated with promising results. This algorithm is one of the algorithms that will be tested and run through our benchmark in order to see how it performs under our conditions

The 48-core Single-chip Cloud Computer (SCC) provides a platform for implementation and capabilities for power management. Previous work done by Shi Sha et al. and Pollawat Thanarungroj et al. has researched the energy and power efficiency of different modes of operation on the SCC in addition to the relation between power consumption, execution time and the number of cores executing the task [10][11].

In this paper we will go over the power management capabilities of the SCC and how they are used in a power management program. Power management algorithms were implemented on the SCC built to use the power management capabilities that are present on the SCC. Finally, the performance of each algorithm is analyzed in order to determine their effectiveness.

II. The Intel Single-Chip Cloud Computer

A. Architecture

The Single-Chip Cloud computer is a 48-core research chip developed by Intel [5]. The chip contains advanced features, such as the ability for fast communication between cores and

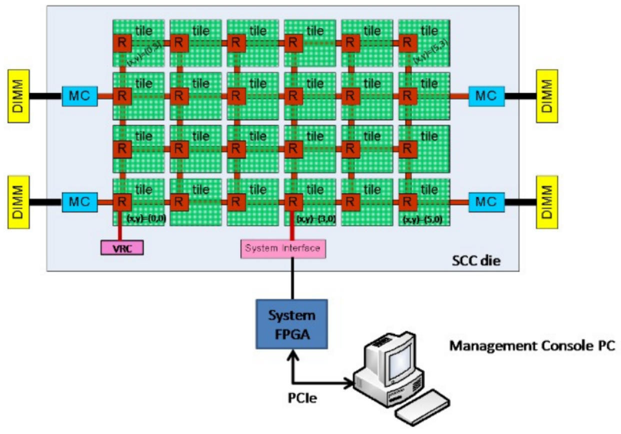

Fig. 1. The Single-chip Cloud Computer Layout [5].

the ability to alternate the frequency and voltage level of different parts of the chip. We also have the capability of reading the exact amount of instructions executed as well as the total number of cycles executed by the core. These values can be read through the model specific registers. These features allow us to implement and analyze different power management algorithms and measure their effectiveness on this platform. The SCC is built out of tiles containing two Intel P54C processors laid out in a six by 4 grid totaling 48 cores as shown in Figure 1. Communication between the tiles is enabled by routers at each tile that connect the cores to the network. This network allows cores to communicate with each other and the four Memory Controllers (MC) that connect the cores to the Dual In-line Memory Modules (DIMM). [4]

An off-chip FPGA module is used to control the chip, such as booting operating systems or loading programs. The FPGA module uses the System Interface to connect to the SCC. Access to the module is granted through a Management Console PC (MCPC). This same module also gives us to the ability to analyze power consumption, as we have access to the voltage level and current drainage numbers for the whole chip. Furthermore, these same values can also be read from the cores, which allows our programs on the SCC to monitor power consumption.

B. Power management capabilities

The SCC contains a Voltage regulator controller (VRC) component. Voltage is adjusted in steps of 0.1 volts in the range between 0.7 and 1.1 volts. Frequency is controlled by a frequency divider with values between 2 and 16 operating at a global frequency of 1600 MHz. All but the highest voltage level imposes restrictions on what frequency dividers are accepted. The voltage and frequency levels can be controlled by directly adjusting the values in the registers of the VRC, but this can be dangerous, as faulty values can lead to breaking the chip. In order to prevent this, a dedicated library has been developed to provide these power management functions in addition to other features. [4]

Calling the power management functions alters the power level of the domain after a short delay. Altering the frequency level only takes a couple of clock cycles before taking effect, but altering the voltage level will take a couple milliseconds before taking effect. The program typically requests a new frequency divider, and while it is possible to change frequency without altering voltage if the voltage level is sufficient, it is more typical to adjust the voltage level so that it is the minimum required voltage level for the new frequency divider. This means that voltage and frequency are typically adjusted in tandem, changing one will cause the other to react as well.

The tiles of the SCC are divided into six different power domains. Each power domain consists of a two by two grid of tiles. Because the tiles are laid out in a six by four grid, the domains are laid out in two rows; the top and bottom half each contains three power domains. One of the cores in the power domain is called the power domain master; only it is capable of calling the power management functions successfully. Should any core other than the power manager attempt to utilize these functions, they will simply execute without actually having any effect on the state of the domain.

C. RCCE-library

With this library we can utilize the advanced and proprietary features of the platform. Some of these features are functions, such as using the MPB or power management functions, but it also provides other ”background maintenance”, such as cache coherency because the SCC does not maintain its own cache coherency protocol. The library provides a basic interface as well as a advanced interface for advanced users. The advanced interface has a more detailed interface that allows for greater control over the functions of the library. [3]

The library also contains an interface for shared memory initialization in case the cores require shared memory with one another or the MPB is insufficient in size or not practical. Though when utilizing shared memory, it is important to properly synchronize the access in order to avoid inconsistencies.

The RCCE-library advanced interface also contains the power management functions used to handle all dynamic voltage and frequency scaling. The power domain master of each power domain can call these functions that provide a safe way of writing into the registers of the VRC. With these functions we can implement different power management algorithms and test their effectiveness on the platform.

III. Architectural Power Management Algorithms

Power management algorithms read load values from the cores and take advantage of intermittent idle time in the cores that enable us to lower the frequency level of the core while having a minimal effect on the execution time of the program running on the core. The implementation of each of these algorithms is explained in the later parts of this paper, here their approach and behaviour is introduced.

A. Core-pool

IBM has developed a power management algorithm for multicore systems that scales well as the number of cores increases [6]. In addition, the algorithm has the ability to take into account idle cores. The algorithm functions by filtering out idle cores, and then attempting to separate the remaining cores into two pools, cores that are active and cores that have slack. The ratio between the numbers of cores in the pools is then used to make power management decisions.

Core-Pool-algorithm reads the current load in percentage as a decision metric. The algorithm has two thresholds according to which each core is categorized. Should the load value be above a certain activity threshold, it is added into a pool of active cores. Afterwards, the load value is tested against a slack threshold. Slack has the meaning that the core is above the activity threshold, but the core still has so much idle time that it can tolerate a lower clock frequency. Should the core be above the activity threshold but below the slack threshold, it is added in the pool of cores with slack. Should the load be below the activity threshold, it is not added into the computation. This process is repeated for each core in the domain.

When all cores in the domain have been processed, the active and slack counters are used to calculate a slack ratio by dividing the slack counter with the active counter. This slack ratio is then used as a combined metric of load for all the cores. The more cores that are in the slack region, the closer will slack ratio be to 1. On the other hand, the more cores that are above the slack threshold, the closer will slack ratio be to zero. The slack ratio will then be tested against an upper threshold and a lower threshold. If slack ratio is between the upper threshold and 1, will frequency be decreased and if the slack ratio is between zero and the lower threshold, will frequency be increased. If the slack ratio is somewhere between the upper and lower threshold, will the frequency remain the same.

More information on how IBM deployed this algorithm and the practical results discovered can be found in [6].

B. PAST

PAST is a power management algorithm that is based on one assumption; future load can be predicted by the previous load. This means that the load in the previous interval is used to determine the appropriate frequency and voltage level for the next interval. [7]

The load in this interval will be measured first. The Million of Instruction per Second (MIPS) value for each core is read and then tested for a small idle threshold. This is used to not include any idle cores when considering what frequency divider is appropriate. After all active cores are known, they are averaged in order to get a load metric for the whole domain. This load metric is then designated to a corresponding load region. Depending on what load region the current load fits into, the frequency and voltage levels are changed to correspond to match it. The same process is repeated at the next measurement interval.

C. Derivative

Derivative is an algorithm that was developed in this research. First, all load values are read from the model specific registers in the cores and we filter out all idle cores. After all idle cores are filtered out, we average the load value so that we get an average MIPS value that represents the activity level in the domain. MIPS values are used to determine whether the power level will be altered. Comparing percentile load values between two different power levels can lead to undesired activity. If all cores are idle, the frequency and voltage levels are put at the lowest possible.

If there are cores that are not idle, the load value from this interval is compared to the previous load in the previous interval to determine if load has increased or decreased. Should the current load be higher in comparison to the last load, the frequency level will be increased. On the other hand, if the current load is lower than the last load, will the frequency level be decreased. Afterwards, the current load is written into memory as it will be needed in the next interval.

IV. Deployment

The SCC is used to serve as a platform for the power management program. The program was written using the Intel C compiler and the RCCE-library.

A. Power Management Program

The program separates the power domain master from the rest of the cores in the domain. The power domain masters read the load values from the other cores in the domain and reacts to them according to the power management algorithm in use.

Accessing load values is achieved through programming the Model Specific Registers (MSR). With the Intel P54C MSR's, we can read the total number of executed instructions as well as the total number of cycles the core has executed. These, along with accurate timer measurements from an off-board FPGA, can be used to calculate the load in percentages, in addition to calculating the MIPS value. An off-board FPGA is mandatory for accurate time measurements because our Dynamic Voltage and Frequency Scaling (DVFS) renders all timer measurements done by our core to be unreliable. This is due to the fact that the timers in cores fail to take in to account the alternating frequency level and thus fail to provide accurate timer measurements. With load values we get a load measurement that is always relative to the domain frequency, while analyzing the MIPS values gives us an absolute value of the domain load through different power domains. Of the 8 cores that belong in each power domain, 7 cores, that are known as the power domain slaves, perform the previously mentioned tasks and write the results in a shared memory buffer. The power domain master does not execute the previously mentioned tasks. The power domain master reads the load values from the shared memory buffer. Now that the power domain master has all the load metrics from each core in the domain, it is left up to the power domain master to analyze these metrics and behave accordingly.

If the whole power domain is not in use, the algorithms are prepared to handle this. All algorithms are scaled to operate even in power domain sizes under 7, and the size of the power domain can be determined at runtime.

In addition to this, there is one designated core that measures the power consumption at steady intervals. This is done by multiplying the voltage and current consumption at any given moment in order to calculate the power consumption. These values represent the power consumption of the whole chip. By calculating the power consumption at fixed intervals and adjusting it so that power consumption is only read when at least some load is present on any one core, we can accurately measure the duration and average power consumption of the program running with the power management algorithm. This allows us to analyze the effectiveness and refine the algorithms for future runs. The interval value is expanded later in this paper.

B. Implementation of Algorithms

The previously introduced power management algorithms have been implemented on the SCC. The description of these algorithms left a lot of details unspecified, many variables had to be tried and tested to find out values that work on the SCC. The SCC frequency divider does allow the frequency to be scaled between values of 2 and 8. The global frequency is set at 1600 MHz, which gives us a maximum frequency of 800 MHz and a minimum frequency of 200 MHz. The frequency divider of the SCC does go down to 16, down to a frequency of 100MHz, but as the lowest frequency dividers between 9 and 16 offer little change in frequency and power consumption, and climbing up from the lowest frequency divider of 16 would lead to our implementations being slow to respond to changes in load. The maximum and minimum level for the frequency divider where inserted into the program by defining the constants MAX_FREQ_DIV_LEVEL as 8 and MIN_FREQ_DIV_LEVEL as 2. In addition, some implementations use the variable powerLevel that is set as the MAX_FREQ_DIV_LEVEL at the start of the program. The function powerChange(int newFrequencyDivider) is called by all algorithms and it sets the frequency level as the one specified by newFrequencyDivider and adjusts the voltage to the minimum value that is accepted by the new frequency divider.

C. Implementation of Core-Pool

The implementation of this algorithm is illustrated in Algorithm 1. First the power domain master reads the percentile load values of the other cores in the domain. Percentile load was chosen in order to easily manage all load metrics from different power levels. Handling MIPS values would have been highly impractical, as we would have to handle load values differently in each power level. Each load is then compared against the constant ACTIVE_THRESHOLD, which has been set at a low value of 5. These values has been chosen, as typical the benchmarks constantly have load that is above this value and when no program is running, the threshold is exceeded only in special circumstances. Should the load be this high, the counter for active cores in increased and the load is tested against the constant SLACK_THRESHOLD which has been set at a value of 17. The reason for this value is partly because the load generated by our benchmark programs is

Algorithm 1 Core-Pool implementation

```
$i \leftarrow 0, activeCount \leftarrow 0, slackCount \leftarrow 0$
while $i \neq domainSize$ do
    $load \leftarrow bufferWithLoads[i]$        # Reads load in percentages
    if $load \geq ACTIVE\_THRESHOLD$ then
        $activeCount++$
        if $load < SLACK\_THRESHOLD$ then
            $slackCount++$
        end if
    end if
    $i++$
end while
if $activeCount > 0$ then
    $slackRatio \leftarrow slackCount/activeCount$
    if $slackRatio > SLACK\_RATIO\_UPPER$ then
        if $powerLevel \neq MAX\_FREQ\_DIV\_LEVEL$ then
            $powerLevel++$        # Frequency div. is increased
            $powerChange(powerLevel)$
        end if
    else if $slackRatio < SLACK\_RATIO\_LOWER$ then
        if $powerLevel \neq MIN\_FREQ\_DIV\_LEVEL$ then
            $powerLevel--$        # Frequency div. is decreased
            $powerChange(powerLevel)$
        end if
    end if
else
    $powerLevel = MAX\_FREQ\_DIV\_LEVEL$
    $powerChange(powerLevel)$
end if
return $b$
```

quite low and partly because it has shown positive results in our testing environment. Should the load be lower than this, the core is marked as having slack by increasing the counter for cores with slack. Once every core in the domain is processed and these two values have been computed, the counter for slack cores is divided by the counter for the cores that are active. Should the resulting value be close to one, it means that most or all of the active cores in the domain have low load. On the other hand, if the resulting value is close to zero, it means that most of the cores in the domain are above the slack threshold, and the frequency should be increased. The constant SLACK_RATIO_UPPER is a value that was set at 0.6. This value was chosen because in a full power domain we have seven cores in computation, which translates to 5 cores needing to have slack for the power level to decrease. This enforces the power domain to stay in higher frequency for a longer time, even if only a few cores have high load. On the other hand the constant SLACK_RATIO_LOWER has been set at 0.4, with the same idea that we can easily react to alternating load values to find a correct power level while disallowing a single core to dictate the power level of the domain. With this threshold values, 5 cores need to be above the slack threshold in a 7 core domain for power level to increase.

On the other hand, if the number of active cores is zero, we will simply go to the lowest power level .

D. Implementation of PAST

The Implementation of this algorithm can be seen in Algorithm 2. Each power domain master reads the MIPS value from each core in the power domain. MIPS give us an ability to use a unified way of measuring load through across all

Algorithm 2 PAST implementation

```
$i \leftarrow 0, numberOfSamples \leftarrow 0, currentLoad \leftarrow 0$
while $i \neq domainSize$ do
    $currentLoad += loadBuffer[i]$
    $numberOfSamples++$
    $i++$
end while
$averageLoad \leftarrow currentLoad/numberOfSamples$
if $currentLoad < 10$ then
    $powerChange(8)$                    # Frequency div. is 8 etc.
else if $currentLoad < 30$ then
    $powerChange(7)$
else if $currentLoad < 50$ then
    $powerChange(6)$
else if $currentLoad < 70$ then
    $powerChange(5)$
else if $currentLoad < 90$ then
    $powerChange(4)$
else if $currentLoad < 110$ then
    $powerChange(3)$
else
    $powerChange(2)$
end if
```

power domains. The MIPS-number is calculated by dividing the number of millions of instructions with the time elapsed since the last measurement interval. The time since the last interval is calculated through the timers of the off board FPGA. With these values the power domain master can calculate the average load in the power domain and adjust power levels accordingly. Each MIPS region is mapped to one power level, and the power domain master will simply switch into the appropriate power level until the next measurement interval. Rapid changes in load will cause PAST to rapidly switch to a power level that is appropriate for the current load.

E. implementation of Derivate

Algorithm 3 Derivate implementation

```
$numberOfSamples \leftarrow 0, i \leftarrow 0, currentLoad \leftarrow 0$
while $i < domainSize$ do
    $coreLoad \leftarrow loadBuffer[i]$          # Reads load in MIPS
    if $load > ACTIVITY\_THRESHOLD$ then
        $numberOfSamples++$
        $currentLoad \leftarrow currentLoad + coreLoad$
    end if
    $i++$
end while
if $numberOfSamples > 0$ then
    $currentLoad \leftarrow currentLoad/numberOfSamples$
    if $currentLoad > lastLoad$ then
        $powerLevel--$                      # Frequency div. is decresed
        $powerChange(powerLevel)$
    else if $currentLoad < lastLoad$ then
        $powerLevel++$                      # Frequency div. is increased
        $powerChange(powerLevel)$
    end if
    $lastLoad \leftarrow currentLoad$
else
    $powerLevel \leftarrow MIN\_FREQ\_DIV\_LEVEL$
    $powerChange(powerLevel)$
end if
```

The implementation of this algorithm is shown in Algorithm 3. Each power domain master first reads the MIPS load values of each core to determine which cores are inactive. The inactivity threshold is represented by the constant ACTIVITY_THRESHOLD with a value of 5. The reasoning for this value is the same as in the previous algorithm, it is enough to filter out cores with idle load and when a program is active, this value is typically exceeded. Any cores that contain a load value higher than this threshold have their load value added into a load buffer. Should no cores be above the minimum activity threshold, the power domain will be set to the lowest power level. Afterwards this load buffer is divided with the amount of active cores in order to determine the average load in the power domain. This load value is then compared against the load value in the previous interval. Should the load value be higher in this interval, the domain power level is increased. If the current load value is below the load value in the previous interval, will the domain power level be decreased. In both of these cases, the power level change will increase or decrease the frequency divider of the power domain, unless it is already at the highest or lower possible power level.

Finally the current load level is written into a buffer that will be used in the next measurement interval.

V. Results

Benchmarks created by Nasa Advanced Supercomputing (NAS) division were used in order the test the effectiveness of each algorithm. In particular the NAS BT.A.25 was used, as it provides a good amount of cores to work with and a good execution time that is not too short or too long. The average execution time and power consumption were recorded across multiple runs on each power management algorithm. The analysis of power consumption and execution time was done by an individual core reading load values and determining when the program starts and stops. Power consumption is read from the registers of the off-board FPGA at a fixed interval. By default, the power consumption measurement is done at a slowish speed of about 1 second, which had to be then shortened in order to better analyze the effectiveness of each algorithm, as the power level can increase or decrease in a short time interval. The shortening of the power consumption measurement interval did cause some spikes in current drainage or voltage to appear when accessing these values, which then have to be discarded. A 0.2 second interval has been chosen between power level measurements to give us a short measurement interval and cause only a minimal amount of spikes in our measurements.

After the program is completed, the core reports the average power consumption while the program was running as well as the execution time of the program. These values are used to analyze the effectiveness of each algorithm on different benchmarks. The benchmark was also run separately at the highest possible frequency and lowest possible frequency in order to better put our research into perspective.

A. NAS BT.A.25 Benchmark

In Table I we can see the effectiveness of each algorithm by comparing the algorithms power consumption and execution time values with each other and the default values of having the program run at a constant 800MHz and 200MHz, which are the maximum and minimum frequency values the algorithms can apply. These results show us that, with these

TABLE I
POWER CONSUMPTION(PC) FIGURES ON NAS BT.A.25

Benchmark	Exec. Time	AVG PC	Total PC
Constant 800MHz	167 Seconds	89 Watts	14.9 KJoules
Core-Pool	252 Seconds	43.3 Watts	10.9KJoules
PAST	293 Seconds	38.2 Watts	11.2KJoules
Derivate	285 Seconds	50.6 Watts	14,5KJoules
Constant 200MHz	387 Seconds	35.8 Watts	13.5KJoules

TABLE II
AMOUNT OF TIME SPENT IN EACH POWER LEVEL ON NAS BT.A.25

Benchmark	800MHz	533MHz	400MHz	320MHz
Core-Pool	3.3 %	36.0 %	43.9 %	15.0 %
PAST	1.2 %	0.0 %	1.0 %	51.8 %
Derivate	35.7 %	21.2 %	13.5 %	7.5 %

Benchmark	266MHz	228MHz	200MHz
Core-Pool	1.8 %	0.0 %	0.0 %
PAST	14.2 %	1.8 %	0.7 %
Derivate	7.8 %	8.4 %	5.8 %

parameters and benchmark load profile, Core-Pool gives the best average between power consumption and runtime. PAST also shows promising results, being closely behind Core-Pool in total power consumption with a lesser average power consumption but a longer execution time. On the other hand Derivate performs the worst of our algorithms by being only slightly more efficient than having the program simply run at maximum power level and also being clearly more inefficient than having the program run at the lowest power level.

In addition, Table II shows how much time each algorithm spent in each power domain during the benchmark. The table shows that the algorithms all behave in a unique way. Core-Pool spends most of its time in the upper values of the power level range, but only rarely spends time in the highest power level. On the other hand PAST spends most of the benchmark in the middle of the power level range. By comparing the results from this table and the execution times in Table I we can see how Core-Pools behaviour to spend time in higher power levels than PAST directly translates to a faster execution time. Furthermore, the differences between the power consumption and execution time of Core-Pool and PAST become clear when cross-referencing with this table.

The inefficiency of Derivate is not solved by the results in Table II. From the table, we can see how the algorithm favours the highest power level, while the execution time is slower than in Core-Pool and almost as slow as PAST. This shows that the overall behaviour of Derivate is in need of rethinking and needs to be severely altered in order for the performance to be up-to-par with the other algorithms.

The performance figures that NAS.BT.A.25 show a clear trend in power management algorithm efficiency, and this same trend has been replicated in other NAS benchmarks. To our knowledge, no other benchmarks have been imported to the SCC-platform or they have an execution time that is too short for our research methods. The amount of research done on these other NAS benchmarks is not enough to show any results, but the same pattern as displayed on BT.A.25 has emerged quite early on these other benchmarks as well.

VI. CONCLUSION

These results show that power management algorithms on multicore systems can be effective, and the effect they have on power consumption is noticeable. In multicore environments where power consumption is an issue, power management algorithms can be used to help solve these problems.

In our test with the selected parameters for each algorithm, Core-Pool shows the best results in overall power efficiency of our algorithms with a slight margin. The improvement in power consumption in comparison to the other benchmarks shows that the algorithm is an effective power management method. PAST comes close behind Core-Pool in efficiency, but the results it shows might be improved by further fine-tuning the parameters to better fit this benchmark.

Derivate on the other hand is clearly in need of redesign and fine-tuning before it shows results that are comparable to the other two algorithms. The power consumption is only slightly improved and the execution time is long compared to how much time it spends in the higher power levels.

REFERENCES

[1] A. Chandrakasan and R. Brodersen, Low Power Digital CMOS Design. Norwell, MA: Kluwer, 1995.

[2] J. M. Rabaey and M. Pedram, Eds., Low Power Design Methodologies. Norwell, MA: Kluwer, 1996.

[3] Intel Labs,SCC External Architecture Specification (EAS) Revision 1.1, Intel, November. 2010.

[4] Intel Labs,SCC Programmer's Guide Revision 1.0, Intel, January. 2012.

[5] Intel Labs,SCC Platform Overview Revision 0.80, Intel, January. 2012.

[6] Karthick Rajamani, Freeman Rawson, Malcolm Ware, Heather Hanson, John Carter, Todd Rosedahl, Andrew Geissler, Guillermo Silva, Hong Hua, Power-Performance Management on an IBM POWER7 Server, IBM, August. 2010.

[7] Stefanos Kaxiras, Margaret Martonosi, *Computer architecture techniques for power efficiency*, Morgan And Claypool, 2008. pp. 27-28.

[8] Borkar, S., *Thousand Core Chips - A Technology Perspective*, 44th ACM/IEEE Design Automation Conference, 2007, pp.746-749, June 2007.

[9] Canturk Isci; Alper Buyuktosunoglu; Chen-Yong Cher; Pradip Bose; Margaret Martonosi; , *An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget*, Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on, pp.347-358, Dec. 2006

[10] Shi Sha; Jiawei Zhou; Chen Liu; Gang Quan; "Power and Energy Analysis on Intel Single-chip Cloud Computer System", Department of Electrical and Computer Engineering, Florida International University, 2012.

[11] Pollawat Thanarungroj; Chen Liu; "Power and Energy Consumption Analysis on Intel SCC Many-Core System", Department of Electrical and Computer Engineering Florida International University, 2011.