Optimization Strategies for Hardware-Based Cofactorization

Daniel Loebenberger[1] and Jens Putzka[2]

[1] b-it
D-53113 Bonn
daniel@bit.uni-bonn.de
http://www.b-it-center.de
[2] MPI für Mathematik
D-53111 Bonn
putzka@mpim-bonn.mpg.de
http://www.mpim-bonn.mpg.de

Abstract. We use the specific structure of the inputs to the cofactorization step in the general number field sieve (GNFS) in order to optimize the runtime for the cofactorization step on a hardware cluster. An optimal distribution of bitlength-specific ECM modules is proposed and compared to existing ones. With our optimizations we obtain a speedup between 17% and 33% of the cofactorization step of the GNFS when compared to the runtime of an unoptimized cluster.

Keywords: General Number Field Sieve (GNFS), Elliptic Curve Method (ECM), hardware cluster, cofactorization step.

1 Introduction

Factoring natural numbers using the elliptic curve method (ECM) is based on the seminal work of Hendrik Lenstra (Lenstra 1987), which is a natural adaption of Pollard's $(p-1)$-method (Pollard 1974) to elliptic curves. In recent implementations of the general number field sieve (GNFS), the ECM is used to factor intermediate sieving results (this is the so called cofactorization step). For example in the record factorization of Franke & Kleinjung (2005) the sieving step produced intermediate numbers of length up to 128 bits. Adapting this to the factorization problem of the number RSA-768 (RSA Laboratories 2007) results in the task of factoring roughly $2 \cdot 10^{12}$ numbers of length up to 140 bit using the ECM.

Since cofactorization is a costly part of the GNFS, it is natural to think about highly specialized hardware realizations of this step, to improve the performance of the GNFS considerably. In particular, since the task consists of many very similar steps, a realization as a hardware *cluster* is suitable. On such a cluster one has many computational units running in parallel that are able to process inputs up to a certain bitlength. The question remains how many of those bitlength-specific modules should be implemented, regardless of the concrete implementation of the corresponding ECM modules. A straightforward approach would be to construct only modules capable of factoring inputs of any size from the GNFS. It is clear, however, that this approach is a great waste of logical resources and that a detailed study of the bitlength-structure of the inputs to the cofactorization step results in much better performance than the naïve approach. Furthermore we quantify the gain we achieve using our optimized construction and generalize our result to arbitrary clusters.

2 The General Number Field Sieve

In this section we give a brief overview of the GNFS in the version which was used by Franke et al. in their record factorization of RSA-640. The GNFS is asymptotically the best known factorization algorithm for large integers. For a more detailed explanation, see for example Lenstra & Lenstra (1993). In this section we will always consider pairs of object, which are indexed by the variable $i \in \{1, 2\}$.

Polynomial Selection: Find good polynomials $F_i(X, Y)$ (see Kleinjung (2006)).

Sieving: Choose two bounds L_i and two bounds B_i. The task is to find many coprime pairs of integers (a, b) with $b > 0$ such that both $F_i(a, b)$ are L_i-smooth. This means that $F_i(a, b)$ decomposes into prime factors smaller than L_i. These pairs (a, b) are called relations. In general it is more than enough to find $\pi(L_1) + \pi(L_2)$ relations. In practice, however, one takes usually some more. We can write for each pair (a, b)

$$F_i(a, b) = R_i(a, b)S_i(a, b)$$

where $R_i(a, b)$ is B_i-rough, i.e. has no factor $< B_i$ and $S_i(a, b)$ is B_i-smooth.

Sieve: Approximation of $\log R_i(a, b)$. This can be done using a lattice sieve (Franke & Kleinjung 2006).

Find Candidates: Take $(R_1(a, b), R_2(a, b))$ for pairs (a, b) if the approximately computed $\log R_i(a, b)$ are below a given bound. These pairs are called candidates. Remove the remaining ones.

Trial Division: For all candidates find the $S_i(a, b)$ (using trial divisions) and calculate the $R_i(a, b)$.

Remove Candidate: If $R_i(a, b) > L_i$ do a fast compositeness test and remove the candidate if $R_i(a, b)$ is pseudoprime.

Apply Strategy: One can precompute a list with pairs of bitlengths which have the property that integers of that size can be factorized in the next step with high probability. For example pairs where both $R_i(a, b)$ are large in some sense can be removed (Kleinjung 2004).

Cofactorization: Find the factors of $R_i(a, b)$ using ECM or MPQS (see for example Cohen (1997)). In our case this should be done using a hardware cluster which uses ECM to find the factors.

Simplification: The relations define a sparse matrix. One now uses some elementary column/row transformations to reduce the size.

Linear Algebra: Solve the resulting system of linear equations.

Computing Square Roots: To be able to find the factor one needs to calculate a square root in a number field.

3 Modelling the Cluster System

Our goal is a model of a hardware cluster (e.g. a COPACOBANA, see Kumar *et al.* (2006), using Virtex4 XC4VSX35 FPGAs). In our specific example the cluster has 16 slots, each containing 8 FPGAs (in the following called chips). Each chip can run several ECM-processes in parallel depending on the size of the corresponding ECM-module. We assume that each chip can only be filled with ECM modules of a particular size. This requirement is from a theoretical point of view unnecessary, but for the concrete realization we have in mind we actually have to require this, since the device controlling all the chips is in our case not able to perform otherwise. Of course modules constructed for a given bitlength can also factor shorter integers. If one wants to factor a number using the cluster, the number is forwarded to a module suitable for its bitlength. The corresponding module then attempts to find a nontrivial factor of the input number. If this succeeds after a certain number of trials (each being a separate run of the ECM with a different elliptic curve), the factor is sent back to the controlling host computer, otherwise the number is discarded. If the factor that is sent back or the remaining cofactor is still composite, another factoring attempt is made. We assume for our estimates that the effort for these additional factorizations is negligible when compared to the first factorization attempt.

The first question we have to answer is the following: From an engineering point of view it is unrealistic to build arbitrary sized ECM modules. What is the smallest bitlength $g \in \mathbb{N}$ for which such a construction is practical? We call this g the *granularity* of the implementation. Of course one cannot give a general answer to this question. The answer heavily depends on the type of the chips one is using and the concrete implementation one has in mind. In our example, we will have $g = 17$ due to the design of the Virtex4 XC4VSX35 FPGAs.

Another question is: How can we get rid of modules for which the numbers of integers having that bitlength is very small? In other words if for a particular bitlength there are only very few numbers to factor, it would be better to factor such numbers using modules capable of factoring larger integers. This would ensure that we would not waste any resources on the cluster, resulting in a better runtime of the cofactorization step.

We describe now the model of the cluster: Let N denote the number of chips on the cluster, e.g. $N = 128$ in our concrete example, and let $\mathcal{D}$ denote the set of inputs to the cofactorization step with $M := \#\mathcal{D}$. For $d \in \mathcal{D}$ let $\operatorname{len}(d)$ denote the bitlength of the number d, i.e. $\operatorname{len}(d) := \lfloor \log_2(d) \rfloor + 1$. Each of the input numbers can be handled by specific modules suitable for their bitlength. The size for which the modules are designed is always a multiple of g. We denote by n_i the number of parallel ECM modules for an integer having $i \cdot g$ bits and by c_i the average runtime of such an integer on the corresponding chips. We are now going to model the classes the numbers may fall into. In general, if we are given an interval $\mathcal{I} := [x, y]$ with $x, y \in \mathbb{N}$ and $x \leq y$, a *partition* of $\mathcal{I}$ is a sequence $\mathcal{C} := (\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_k) \in \mathbb{N}^k$ for some $k \in \mathbb{N}$, with $x = \mathcal{C}_0 < \mathcal{C}_1 < \cdots < \mathcal{C}_k = y$. We call k the *size* of the partition $\mathcal{C}$. The interval $(\mathcal{C}_{i-1}, \mathcal{C}_i]$ is called the *i-th subinterval* of $\mathcal{C}$. If now $\mathcal{C}^1$ and $\mathcal{C}^2$ are partitions of $\mathcal{I}$, we say that $\mathcal{C}^2$ is a *refinement* of $\mathcal{C}^1$ if for any $0 \leq i \leq k$ there is some j, such that $\mathcal{C}^1_i = \mathcal{C}^2_j$. In other words that means that we have subdivided the subintervals of $\mathcal{C}^1$ into smaller pieces without changing already existing cuts and we write $\mathcal{C}^1 \preceq \mathcal{C}^2$. Conversely, $\mathcal{C}^1$ is called a *coarsening* of $\mathcal{C}^2$. For our purposes we only consider partitions $\mathcal{C}$ of the interval $\mathcal{I} = [x, y]$ where $x := \lfloor \min(\text{len}(d) \mid d \in \mathcal{D}) \rfloor_g$ and $y := \lceil \max(\text{len}(d) \mid d \in \mathcal{D}) \rceil_g$, where the notation $\lfloor . \rfloor_g$ ($\lceil . \rceil_g$) means that the rounding is done down to (up to) the next multiple of g. Additionally we require that for any $0 \leq i < \#C$ the number $\mathcal{C}_i$ is a multiple of g. We will call such partitions *g-partitions* of the intervall induced by $\mathcal{D}$. In particular the finest partition we will consider is the g-partition $\mathcal{C}^f := (x, x + g, x + 2g, \ldots, y)$ and the possible partitions we may have at the end are always coarsenings of $\mathcal{C}^f$.

For the following, fix a data set $\mathcal{D}$ and define $K := \#\mathcal{C}^f - 1 = (y - x)/g$. Now given any $\mathcal{C} \preceq \mathcal{C}^f$ of size k, let $a_i(\mathcal{C}) \in \mathbb{N}$ be the number of occurrences in the i-th subinterval of $\mathcal{C}$, i.e. $a_i(\mathcal{C}) := \# \{ d \in \mathcal{D} \mid \text{len}(d) \in (\mathcal{C}_{i-1}, \mathcal{C}_i] \}$. For later use we define the input distribution

$$\alpha(\mathcal{C}) := \left(\frac{a_1(\mathcal{C})}{M}, \ldots, \frac{a_k(\mathcal{C})}{M} \right) \in \mathbb{R}^k.$$

If we consider the ith subinterval of $\mathcal{C}$ the average cost of factoring such a number is $c_{\mathcal{C}_i/g}$. The space used for such a module is roughly $1/n_{\mathcal{C}_i/g}$. Thus the area-time product for class i is given by

$$\vartheta_i(\mathcal{C}) := \frac{c_{\mathcal{C}_i/g}}{n_{\mathcal{C}_i/g}}.$$

A *layout* of the cluster is given by an ordered partition $\ell \vdash_k N$ of the N chips into k summands, one for each class. Thus we have

$$\ell \vdash_k N :\Longleftrightarrow \ell = (\ell_1, \ldots, \ell_k) \in \{1, \ldots, N\}^k \wedge \sum_{1 \leq i \leq k} \ell_i = N,$$

with $\ell_i > 0$, implying $N \geq k$. That means we assume that the number of chips is always greater than the number of classes, which is also reasonable. Note that we have indeed two different notions of partitions here: First a partition of an interval and second an additive ordered partition of a natural number. This could of course be unified, but for our work it is preferable to have these two different notions, since for the former notion we emphasize on the variable number of subintervals while for the latter we assume a fixed number of summands.

Write $\mathcal{C}|_j$ for the restriction of $\mathcal{C}$ on its first j subintervals. The minimal runtime for $\mathcal{C}|_j$ is given by

$$\mu_{\mathcal{C}}(N,j) := \min_{\ell \vdash_j N} \max_{1 \le i \le j} \frac{\vartheta_i(\mathcal{C}|_j) \cdot a_i(\mathcal{C}|_j)}{\ell_i} \tag{1}$$

The value $\mu_{\mathcal{C}}(N,j)$ is indeed a time measurement, since c_i is given in seconds, n_i has unit 1/ chip and ℓ_i has unit chip. We will use the following convention: If we write $\mu_{\mathcal{C}}(N)$ we actually mean $\mu_{\mathcal{C}}(N, \#\mathcal{C}-1)$. Further we define

$$\tau(N) := \min_{\mathcal{C} \preceq \mathcal{C}^f} \mu_{\mathcal{C}}(N) \tag{2}$$

Equation (1) and (2) actually depend on the data set $\mathcal{D}$ and we write $\mu_{\mathcal{D},\mathcal{C}}(N,j)$ and $\tau_{\mathcal{D}}(N)$, respectively, if there is more than one data set under consideration. In the following we will show how one can compute $\mu_{\mathcal{C}^f}(N)$ efficiently, namely with $\mathcal{O}(N \cdot K)$ arithmetic operations. Note that the imprecision of considering arithmetic operations only is in our case not a problem, since the size of the numbers is bounded from above by a constant.

We can compute Equation (1) easily using Bellman's dynamic programming. To do so, we need to handle two things:

1. The solutions for the boundaries have to be computed (i.e. for the case $j = 1$):
$$\mu_{\mathcal{C}}(N,1) = \frac{\vartheta_1(\mathcal{C}|_1) \cdot a_1(\mathcal{C}|_1)}{N} \tag{3}$$
2. We need a recursion formula for $\mu_{\mathcal{C}}(N,j)$. Assume we know $\mu_{\mathcal{C}}(N', j-1)$ for all $N' < N$. Then we have
$$\mu_{\mathcal{C}}(N,j) = \min_{N' < N} \max \left(\mu_{\mathcal{C}}(N', j-1), \frac{\vartheta_j(\mathcal{C}|_j) \cdot a_j(\mathcal{C}|_j)}{N - N'} \right) \tag{4}$$

The function $\mu_{\mathcal{C}}(N,j)$ can thus be computed with $\mathcal{O}(N \cdot j)$ arithmetic operations.

Let us now compute the function $\tau(N)$. The total number of classes $\mathcal{C} \preceq \mathcal{C}^f$ is $2^K/4$. Since K will be small in all our examples of the GNFS, a straightforward algorithm would just compute $\mu_{\mathcal{C}}(N)$ for all $\mathcal{C} \preceq \mathcal{C}^f$ and select the classes with minimal runtime. Employing such an algorithm for the computation of $\tau(N)$ will use $\mathcal{O}(NK2^K)$ arithmetic operations.

We will now describe a greedy approach which will find in many cases the optimal classes using only $\mathcal{O}(K)$ evaluations of the function $\mu_{\mathcal{C}}(N)$ for various $\mathcal{C} \preceq \mathcal{C}^f$, i.e. compute $\tau(N)$ with $\mathcal{O}(N \cdot K^2)$ arithmetic operations: Let $\mathcal{C} := [\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_k]$ be any partition of the interval $\mathcal{I} = [x, y]$.

For $p \in [1, K-1]$ denote by $\mathcal{C}^{(p)}$ the refinement of $\mathcal{C}$ at position $g \cdot p$. Our algorithm will work as follows: Starting from the partition (x, y), we successively refine (x, y) until the optimal partition is found. In particular if we are given in step r a partition $\mathcal{C}$, we compute $\mu_{\mathcal{C}^{(p)}}(N)$ for all p and take in the next round the partition $\mathcal{C}^{(p)}$ with the smallest runtime $\mu_{\mathcal{C}^{(p)}(N)}$. If there are two positions p_1, p_2 with the same minimal runtime, we select one of the partitions randomly for the next step. This approach is indeed greedy, since we take in every round the best subdivision. The algorithm terminates if for all p the value $\mu_{\mathcal{C}(p)}(N)$ is not strictly smaller than $\mu_{\mathcal{C}}(N)$. In this case the partition $\mathcal{C}$ is returned. Observe that this algorithm will in general *not* find the optimal classes, since we cannot guarantee that the algorithms terminates in a local minimum. In our experiments, however, this heuristic indeed computed $\tau(N)$ in all our examples.

In order to measure the advantage of our optimization, we compare the estimated runtime of the cluster using our construction with the runtime of a naïvely constructed cluster, i.e. a cluster only containing bitlength-specific modules for numbers having y bits. On such a cluster the runtime for a data set $\mathcal{D}$ of M numbers is bounded from below by the following expression:

$$\sigma^{-}_{\mathcal{D}}(N) := \frac{1}{N \cdot n_K} \sum_{1 \le i \le K} c_i a_i \tag{5}$$

and bounded from above by

$$\sigma^{+}_{\mathcal{D}}(N) := \frac{M c_K}{N n_K} \tag{6}$$

with $K := \#\mathcal{C}^f - 1$ as above. The first estimate is a bit optimistic since the runtime of a module does not only depend on the input but also on the arithmetic built into the module. Further the second estimate is too pessimistic, since a module running on smaller input numbers will also run faster on average.

We use the functions

$$\gamma^{-}_{\mathcal{D}}(N) := \frac{\sigma^{-}_{\mathcal{D}}(N) - \tau_{\mathcal{D}}(N)}{\sigma^{-}_{\mathcal{D}}(N)}$$

and

$$\gamma^{+}_{\mathcal{D}}(N) := \frac{\sigma^{+}_{\mathcal{D}}(N) - \tau_{\mathcal{D}}(N)}{\sigma^{+}_{\mathcal{D}}(N)}$$

as lower and upper bounds, respectively, to measure the runtime gain we achieve with our optimized cluster. This expression is exactly the runtime gain achieved by the optimization (having runtime $\tau_{\mathcal{D}}(N)$) in contrast to the naïvely constructed cluster (having runtime between $\sigma^{-}_{\mathcal{D}}(N)$ and $\sigma^{+}_{\mathcal{D}}(N)$).

4 Concrete Statistical Analyses

We will now perform a rigorous statistical analysis of six concrete runs of the GNFS up to the cofactorizations step for the number RSA-768 using Franke and Kleinjung's implementation, and study the function $\tau(N)$ for these particular inputs: Each data set $\mathcal{D}$ consists of many $(2 \cdot 10^8)$-rough composite numbers of bitlength between 58 and 160, each $\mathcal{D}$ being a specific output of the sieving step of the GNFS for different choices of a polynomial pair and the sieving region of the lattice siever. Following von zur Gathen *et al.* (2007), we estimate the

Table 1. Number of parallel ECM-modules per chip depending on the bitlength

Bitlength $17i$	17	34	51	68	85	102	119	136	153	170
Processes n_i	32	26	22	18	15	12	10	9	8	7

Table 2. Average runtime of the ECM on a Virtex4 XC4VSX35 FPGA

Bitlength $17i$	17	34	51	68	85
Cost c_i (in μs)	491.49125	673.9225	856.35375	1038.785	1221.21625

Bitlength $17i$	102	119	136	153	170
Cost c_i (in μs)	1403.6475	1586.07875	1768.51	1950.94125	2133.3725

Table 3. Relative frequencies of the input data

Bitlength	$0-68$	$69-85$	$86-102$	$103-119$	$120-136$	$137-153$
$\mathcal{D}_1$	0.0015	0.0553	0.4540	0.0886	0.2826	0.1181
$\mathcal{D}_2$	0.0007	0.0547	0.4493	0.0889	0.2823	0.1241
$\mathcal{D}_3$	0.0008	0.0540	0.4533	0.0881	0.2836	0.1203
$\mathcal{D}_4$	0.0009	0.0567	0.4440	0.0874	0.2902	0.1209
$\mathcal{D}_5$	0.0011	0.0518	0.4306	0.0875	0.2992	0.1299
$\mathcal{D}_6$	0.0009	0.0461	0.4340	0.0834	0.3031	0.1326
Mean	0.0010	0.0531	0.4442	0.0873	0.2902	0.1243
Stdev.	0.0003	0.0038	0.0099	0.0020	0.0091	0.0058

number of parallel ECM modules and the runtime on the Virtex4 XC4VSX35 FPGAs according to Table 1 and 2, respectively. In the implementation that was used only modules for $17i$ bit integers were build. Note that such a module will also be capable of factoring samller integers.

Let us have a look at the distribution $\alpha(\mathcal{C}^f)$ of the input data for the various data sets (see Table 3). Note the low standard deviation of the corresponding entries. In Figure 1 a histogram as well as the distribution on the classes $\mathcal{C}^f$ is given for data set $\mathcal{D}_1$.

We now employ our model to find an optimal layout for the cluster and compute the runtime gain we achieved with our optimization. Let the notation be as in Section 3. In the case of the COPACOBANA we will have $N = 8 \cdot 16 = 128$. There are 351306039 ordered partitions of the number 128 in not more than 6 parts. The total number of layouts of the cluster, including the choice of the classes is in our example 402858941.

After having computed the function $\tau_{\mathcal{D}}(128)$ for all data sets $\mathcal{D}$ we obtain for every set an optimal layout (consisting of the interval partition $\mathcal{C}$ and the distribution of chips ℓ). If we take the result of the optimization for data set $\mathcal{D}_1$, for example, we will have 47 modules for integers of up to 102 bit, 58 for integers up to 136 bit and 23 for the remaining integers (up to 153 bit). The size of the first class is in this case 102 bit, the size of the second one 34 bit and of the third class 17 bit. The results are summarized in Table 4 and 5.

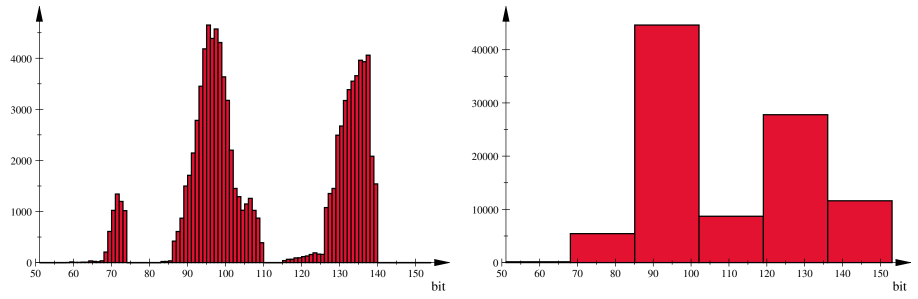


Fig. 1. Left: Histogram of data set $\mathcal{D}_1$. Right: Distribution onto specific modules.

Table 4. Optimal partitions for the data sets $\mathcal{D}_1$, $\mathcal{D}_2$ and $\mathcal{D}_3$

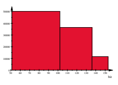
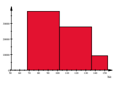
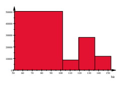
	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$
			
$(\mathcal{C}_{i+1} - \mathcal{C}_i)/g$	(3,2,1)	(1,2,2,1)	(3,1,1,1)
ℓ	(47, 58, 23)	(1, 46, 57, 24)	(48, 11, 45, 24)
$\tau_{\mathcal{D}}$ (μs)	124966.936	96137.13955	126309.5441
$\#\mathcal{D}$	98322	75013	99488
$\tau_{\mathcal{D}}/\#\mathcal{D}$	1.271	1.2816	1.2696

Table 5. Optimal partitions for the data sets $\mathcal{D}_4$, $\mathcal{D}_5$ and $\mathcal{D}_6$

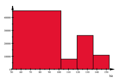
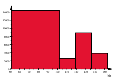
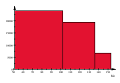
	$\mathcal{D}_4$	$\mathcal{D}_5$	$\mathcal{D}_6$
			
$(\mathcal{C}_{i+1} - \mathcal{C}_i)/g$	(3,1,1,1)	(3,1,1,1)	(3,2,1)
ℓ	(47, 11, 46, 24)	(45, 11, 47, 25)	(44, 59, 25)
$\tau_{\mathcal{D}}$ (μs)	113592.0763	37653.16612	65015.11716
$\#\mathcal{D}$	90141	29719	50273
$\tau_{\mathcal{D}}/\#\mathcal{D}$	1.2602	1.267	1.2932

Table 6. Performance gain for data set $\mathcal{D}_1$ (in percent) of the optimized cluster

	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$	$\mathcal{D}_5$	$\mathcal{D}_6$
$\gamma_{\mathcal{D}}^{-}$	17.47	16.97	17.66	18.38	18.4	16.88
$\gamma_{\mathcal{D}}^{+}$	33.29	32.73	33.36	33.86	33.5	32.12

In order to measure the advantage of our optimization, we use the estimates from Section 3. We have here at maximum 153 bit numbers and use the values in the tables above. The result of our optimization is shown in Table 6.

5 Generalizations to an Arbitrary Number of Clusters

Fix one data set $\mathcal{D}$. In this section we analyze the behaviour of the function $\gamma^-(N)$ for $N \to \infty$.

In practice a growing N would mean that we employ not only one COPACOBANA, but a whole collection of these, running simultaneously, and optimize over the whole set of chips. We will now show that the runtime gain achieved by this collection of clusters converges to roughly 21% when compared to a collection of naïvely constructed clusters. It is clear that the actual gain however will strongly depend on the input data $\mathcal{D}$.

Now let's say we are going to build m clusters and we wish to optimize the number of bitlength specific ECM modules as above. The formulae in Section 3 are still valid, except that we will have $N = 128m$ chips in a collection of m clusters instead of $N = 128$ as above.

We wish to compute $\lim_{N\to\infty} \gamma^{\pm}(N)$. To do so, we first need to compute $\tau(N)$ for $N \to \infty$. Unfortunately, the dynamic programming approach used above is only useful if we consider fixed N, but does not tell us anything about the limit. In Figure 2 the value of $\gamma^-(N)$ is plotted for the case of $m \in \{1, \ldots, 100\}$ clusters using data set $\mathcal{D}_1$. Note that this observation follows also our intuition, since with an increasing number of clusters one cannot expect more runtime gain.

Assume we are given classes $\mathcal{C} \preceq \mathcal{C}^f$. Set $k := \#\mathcal{C} - 1$. In order to be able to compute the limit, we look at the problem of computing $\mu_{\mathcal{C}}(N)$ over the reals, i.e. we will have $\ell \in \mathbb{R}^k$. With this simplifications it is clear that the expression

$$\max_{1\le i\le k} \frac{\vartheta_i(\mathcal{C}) \cdot a_i(\mathcal{C})}{\ell_i}$$

is minimal if and only if

$$\frac{\vartheta_i(\mathcal{C}) \cdot a_i(\mathcal{C})}{\ell_i} = \frac{\vartheta_j(\mathcal{C}) \cdot a_j(\mathcal{C})}{\ell_j} \text{ for all } i, j \in \{1, \ldots, k\}$$

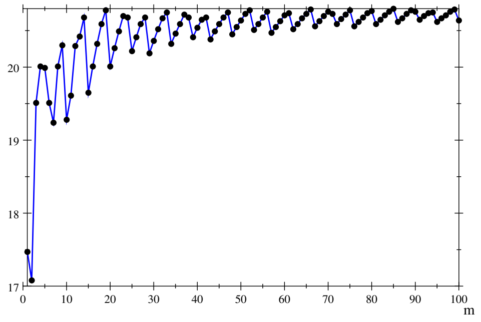


Fig. 2. Lower bound on the runtime gain for an increasing number m of clusters

Write $\vartheta'_i(\mathcal{C}) := \vartheta_i(\mathcal{C}) \cdot a_i(\mathcal{C})$. We end up in solving the following system of equations:

$$\begin{aligned} \ell_1 + \cdots + \ell_k &= N \\ \vartheta'_1(\mathcal{C}) \cdot \ell_2 &= \vartheta'_2(\mathcal{C}) \cdot \ell_1 \\ \vdots \quad & \quad \vdots \\ \vartheta'_1(\mathcal{C}) \cdot \ell_k &= \vartheta'_k(\mathcal{C}) \cdot \ell_1 \end{aligned}$$

This system of k equations is linear in the k unknowns $\ell_1, \ldots, \ell_k$, having the solution

$$\ell_i = \frac{\vartheta'_i(\mathcal{C})N}{\vartheta'_1(\mathcal{C}) + \cdots + \vartheta'_k(\mathcal{C})}$$

We could have used this approach also for our computation of $\mu_{\mathcal{C}}(n)$ in Section 3. There we would have computed the approximate partition of N (being a vector of reals) and would then have rounded the results appropriately. To find the minimum we would have then to round 2^k times resulting in an algorithm that would have used $\mathcal{O}(k \cdot 2^k)$ arithmetic operations, which is of course preferable if k is small compared to N. Back to our question of computing the limit we have

$$\lim_{N\to\infty} \mu_{\mathcal{C}}(N) = \lim_{N\to\infty} \frac{1}{N} \sum_{1\leq i<\#\mathcal{C}} \vartheta'_i(\mathcal{C}) \quad \text{and} \quad \lim_{n\to\infty} \tau(N) = \min_{\mathcal{C}\preceq\mathcal{C}^f} \lim_{N\to\infty} \mu_{\mathcal{C}}(N).$$

Furthermore

$$\lim_{N\to\infty} \sigma^-(N) = \lim_{N\to\infty} \frac{1}{N \cdot n_K} \sum_{1\leq i\leq K} a_i \cdot c_i \quad \text{and} \quad \lim_{N\to\infty} \sigma^+(N) = \lim_{N\to\infty} \frac{Mc_K}{Nn_K}$$

Together

$$\lim_{N\to\infty} \gamma^-(N) = \min_{\mathcal{C}\preceq\mathcal{C}^f} 1 - \frac{n_K \sum_{1\leq i<\#\mathcal{C}} \vartheta'_i(\mathcal{C})}{\sum_{1\leq i\leq K} c_i \cdot a_i}$$

and

$$\lim_{N\to\infty} \gamma^+(N) = \min_{\mathcal{C}\preceq\mathcal{C}^f} 1 - \frac{n_K \sum_{1\leq i<\#\mathcal{C}} \vartheta'_i(\mathcal{C})}{Mc_K}.$$

Table 7 shows the results for our six test sets. We observe again that the corresponding values for the different data sets are very similar. Thus it seems that only the distribution of the inputs is crucial for the outcome of the optimization.

Table 7. Bounds on the limit of the runtime gain (in percent) for the various data sets

	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$	$\mathcal{D}_5$	$\mathcal{D}_6$
$\lim_{N\to\infty} \gamma_{\mathcal{D}}^{-}$	20.81	20.58	20.70	20.56	20.00	19.81
$\lim_{N\to\infty} \gamma_{\mathcal{D}}^{+}$	35.99	35.66	35.82	35.63	34.80	34.51

6 Conclusion

We have described a mathematical model of a hardware cluster like the COPACOBANA. Using this model we were able to compute the optimal distribution of bitlength specific modules on such a cluster efficiently, independent of which concrete ECM implementation was used. For our optimization it is necessary to have an estimate of the expected input distribution. This is in the case of the GNFS a nontrivial question (given some fixed parameter set), but it seems that the outputs of the GNFS always follow a certain distribution. To study this distribution in general is a challenging task and requires a deep understanding of the number theoretical properties of the inputs for the cofactorization step. Results in this direction are reserved for a forthcoming publication. The methods that were used are standard and were well studied in the 1960th and the 1970th. Nonetheless our optimization gives a speedup between 17% and 33% for the cofactorization step of the GNFS. As far as we know such a mathematical optimization was never done before for a hardware cluster like the COPACOBANA. Additionally our results are applicable for any scalable problem, when one wants to implement it efficiently on a dedicated hardware cluster.

Acknowledgements

Both authors were funded by the b-it foundation, the state of Northrhine-Westfalia and the German Federal Office for Information Security (BSI). The second author was additionally funded by the MPI and the Hausdorff-Center for Mathematics in Bonn. We want to express our thanks to Thorsten Kleinjung for helpful discussions on the output of the sieving step. Additional thanks go to Jérémie Detrey.

References

1. Bellman, R.: Dynamic Programming. Princeton University Text (1957)
2. Cohen, H.: A course in computational algebraic number theory. Springer, Berlin (1997)
3. Franke, J., Kleinjung, T.: RSA 640 (2005), `http://www.crypto-world.com/announcements/rsa640.txt`
4. Franke, J., Kleinjung, T.: Continued Fractions and Lattice Sieving (Unpublished) (2006), `http://www.math.uni-bonn.de/people/thor/confrac.ps`
5. von zur Gathen, J., Güneysu, T., Kargl, A., Loebenberger, D., Paar, C., Putzka, J.: Faktorisierung großer Zahlen: Hardware für Elliptische Kurven Faktorisierung. Technical report, HGI Bochum, b-it Bonn & Siemens AG München (2007)

6. Kleinjung, T.: Cofactorisation Strategies for the Number Field Sieve and an Estimate for the Sieving Step for Factoring 1024-bit Integers (Unpublished) (2004), `http://www.math.uni-bonn.de/people/thor/cof.ps`
7. Kleinjung, T.: On Polynomial Selection for the General Number Field Sieve. Mathematics of Computation 75(256), 2037–2047 (2006), `http://dx.doi.org/10.1090/S0025-5718-06-01870-9`
8. Kumar, S., Paar, C., Pelzl, J., Pfeiffer, G., Schimmler, M.: Breaking ciphers with COPACOBANA –A Cost-Optimized Parallel Code Breaker. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 101–118. Springer, Heidelberg (2006), `http://dx.doi.org/10.1007/11894063_9`
9. Lenstra, A.K., Lenstra Jr., H.W. (eds.): The development of the number field sieve. Lecture Notes in Mathematics, vol. 1554. Springer, Berlin (1993)
10. Lenstra Jr., H.W.: Factoring integers with elliptic curves. Annals of Mathematics 126, 649–673 (1987)
11. Pollard, J.M.: Theorems on factorization and primality testing. Proceedings of the Cambridge Philosophical Society 76, 521–528 (1974)
12. RSA Laboratories. The RSA Challenge Numbers (2007)